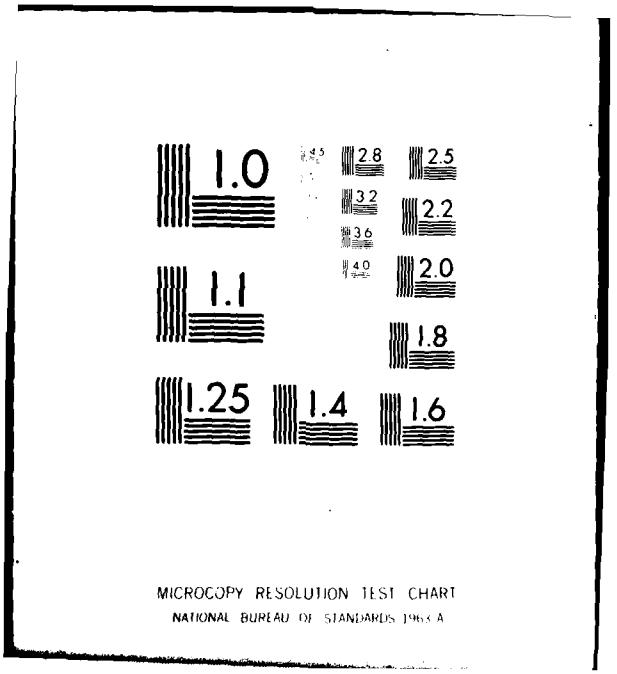
AD-A087 996 COMPUTER CORP OF AMERICA CAMBRIDGE MA F/G 9/2
FUNDAMENTAL ALGORITHMS FOR CONCURRENCY CONTROL IN DISTRIBUTED D--ETC(U)
MAY 80 P A BERNSTEIN, N GOODMAN F30602-79-C-0191
UNCLASSIFIED RADC-TR-80-158 NL

1 OF 3
AD
A087996


1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963 A

RADC-TR-80-158
Final Technical Report
May 1980

LEVEL II

FUNDAMENTAL ALGORITHMS FOR CONCURRENCY CONTROL IN DISTRIBUTED DATABASE SYSTEMS

Computer Corporation of America

Philip A. Berstein
Nathan Goodman

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-80-158 has been reviewed and is approved for publication.

APPROVED: Thomas F. Lawrence

THOMAS F. LAWRENCE
Project Engineer

APPROVED: Alan R Barnum

ALAN R. BARNUM, Assistant Chief
Information Sciences Division

FOR THE COMMANDER: John P. Huss

JOHN P. HUSS
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (ISCP), Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return this copy. Retain or destroy.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

(19) REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

(18) RADC-TR-80-158

2. GOVT ACCESSION NO. AD-A084 996

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)

(6) FUNDAMENTAL ALGORITHMS FOR CONCURRENCY CONTROL IN DISTRIBUTED DATABASE SYSTEMS

(9) 5. TYPE OF REPORT & PERIOD COVERED
Final Technical Report
Jul 1979 - Jan 1980

6. PERFORMING ORG. REPORT NUMBER
N/A

7. AUTHOR(s)

(10) Philip A. Bernstein
Nathan Goodman

8. CONTRACT OR GRANT NUMBER(s)

(15) F30602-79-C-0191

9. PERFORMING ORGANIZATION NAME AND ADDRESS

Computer Corporation of America
575 Technology Square
Cambridge MA 02139

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

62702F
55812111
(12) 2266

11. CONTROLLING OFFICE NAME AND ADDRESS

Rome Air Development Center (ISCP)
Griffiss AFB NY 13441

(11) 12. REPORT DATE
May 1980

13. NUMBER OF PAGES
268

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)

Same (16) 5581, 2530

15. SECURITY CLASS. (of this report)
UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

(17) 21, 01

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Thomas F. Lawrence (ISCP)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Distributed Systems
Databases
Concurrency Control

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report surveys and consolidates the state-of-the-art in distributed database concurrency control. The heart of the analysis is a decomposition of the concurrency control problem into two major sub-problems: read-write and write-write synchronization. The report describes a series of synchronization techniques for solving each sub-problem and shows how to combine synchronization techniques into algorithms that solve the entire concurrency control problem. Algorithms constructed

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

387285

in this way are called concurrency methods. The report describes 48 principle concurrency control methods; these methods include all practical algorithms for distributed database concurrency control that have appeared in the literature plus several new algorithms.

In addition, an analysis of principal concurrency control methods in qualitative terms is performed. The analysis considers four cost factors: communication overhead, local processing overhead, transaction restarts, and transaction blocking. The results indicate that only about 10 of the principal concurrency control methods are reasonable choices in practice. This list of so-called dominant methods includes both old and new algorithms.

This report concludes with recommendations for future research directions.

Technical Summary

The long range goal of research on distributed database concurrency control is to develop a methodology for designing good concurrency control methods for a given system environment and a given class of applications. As a first step toward this goal, this study presents an overview of the state-of-the-art, of distributed database concurrency control.

We set the stage for this study by presenting a model of transaction processing in a distributed database system, emphasizing the essential inter-site interactions needed to process users' transactions. This model provides a common framework for describing and analyzing concurrency control methods, a framework that has been lacking in the literature.

We then review the mathematical theory of concurrency control. We formalize the principal correctness criterion for a concurrency control method -- namely serializability. And we show how the overall problem of attaining serializability can be decomposed into two sub-problems -- read-write and write-write synchronization. This decomposition is the cornerstone of our paradigm for the design and analysis of concurrency control methods.

We exploit this paradigm to examine fundamental read-write and write-write synchronization techniques outside the context of

any specific concurrency control method. We consider virtually all known synchronization techniques and show that these techniques can be understood as variations of two basic techniques -- two-phase locking (2PL) and timestamp ordering (T/O). This consolidation of the state-of-the-art is possible in large part because of our read-write/write-write paradigm for concurrency control and our common transaction processing model introduced earlier.

We study the space of concurrency control methods that can be constructed from the previously described techniques. We show this space to be enormous: there are thousands of ways of combining synchronization techniques into complete concurrency control methods. However, we identify 48 of these methods as principal methods. Most of these principal methods have not been described in the literature previously, and several of these new methods have interesting performance characteristics.

Next, we analyze the performance of principal methods. We describe the four main performance measures for concurrency control methods -- communication overhead, local processing overhead, transaction restarts, and transaction blocking. We analyze the major synchronization techniques and the principal concurrency control methods relative to each performance measure in qualitative terms. We show that no method has optimal performance under all four measures, which means that no method

can be expected to perform optimally for all system environments and applications. However, we identify 11 methods as dominant methods -- for any given system and application we believe that one of these 11 dominant methods will be optimal. In addition, we suggest a design scenario for choosing among dominant methods for certain kinds of stereotypical applications.

In the Appendix we discuss three methods that have appeared in the literature but do not fit into our framework of techniques and methods. While these methods are intellectually impressive (and, in some cases, famous), none is of practical significance in a distributed database environment.

Briefly, then, the state-of-the-art in distributed database concurrency control is as follows.

1. a large number of correct methods for distributed database concurrency control are known;
2. many important characteristics of system environments and applications have been isolated;
3. the relative performance of concurrency control methods has been qualitatively analyzed for some combinations of system and application characteristics.

We close with recommendations on profitable avenues for future research.

Table of Contents

EVALUATION

This final report provides the basis for the structuring of new algorithms for Concurrency Control (CC) and provides the basis for codifying engineering information for the design of distributed databases which are particularly relevant to Command and Control Systems.

This effort applies to TPO-R3, Thrust D, "C^2 Information Processing"; Subthrust 1, "C^2 Information System Structures; specifically, Project 5581, Task 21, "Network and Distributed Processing Studies" and Project 2530, Task 01, "Distributed Data Processing".

This report forms the basis for the follow-on effort entitled "DDB Control and Allocation" in which CC algorithms will be evaluated and a System Designer's Handbook will be developed.

Thomas F. Lawrence

THOMAS F. LAWRENCE
Project Engineer

1. The Concurrency Control Problem

1.1 Introduction

Concurrency control is a necessary component of any multi-user database management system (DBMS). Its role is to coordinate the database interactions of users who are accessing a database at the same time. Concurrency control permits multiple users to access a database in a multi-programmed fashion while preserving the illusion that each user is executing alone on a dedicated system. The main technical difficulty in attaining this goal is to prevent database updates performed by one user from interfering with database retrievals and updates performed by another. The nature of this problem is illustrated in Section 1.2.

The concurrency control problem is exacerbated in a distributed database management system (DDBMS) for two reasons. First, users may access data stored in many different computers in a distributed system. And second, a concurrency control mechanism at one computer cannot instantaneously know about interactions at other computers.

Concurrency control has been an active research and development field for the past several years, and the problem for non-distributed DBMSs is well in hand. One approach, called two-phase locking, has been accepted as a de facto standard, and a broad mathematical theory, called serializability theory, has been developed to analyze the problem. Current research on non-distributed concurrency control is focused on evolutionary improvements to two-phase locking, detailed performance analysis and optimization, and extensions to serializability theory.

Distributed concurrency control, by contrast, is in a state of extreme turbulence. More then twenty concurrency control algorithms have been proposed for DDBMSs, and several have been, or are being, implemented. These algorithms are usually complex and hard to understand. It is difficult to prove the correctness of these algorithms, and indeed many are incorrect for technical reasons. The algorithms are not described in standard terminology and different algorithms make different assumptions regarding the underlying DDBMS environment. For these reasons, it is difficult to compare the many proposed algorithms, even in qualitative and general terms. Naturally each author proclaims his approach to be superior, but there is little compelling evidence to support any such claims.

The purpose of this report is to survey and consolidate the state-of-the-art in DDBMS concurrency control. We shall

introduce a standard terminology for describing DDBMS concurrency control algorithms and a standard model for the DDBMS environment. Using this terminology and model as a foundation, we are able to decompose the overall concurrency control problem into two major sub-problems, called read-write and write-write synchronization. Every concurrency control algorithm must include a sub-algorithm for solving each of these sub-problems. To understand and analyze an overall concurrency control algorithm, a useful first step is to isolate the sub-algorithms employed for each sub-problem.

Despite the large number of proposed algorithms, we will see that relatively few sub-algorithms are possible. Indeed, we will see that the sub-algorithms used by all practical DDBMS concurrency control algorithms are variations of two basic techniques: two-phase locking, and a technique called timestamp ordering. Thus, the state-of-the-art in DDBMS concurrency control is far more coherent than a review of the literature would indicate.

Having structured the problem of DDBMS concurrency control in this way, several benefits accrue.

1. We are able to describe proposed algorithms in a common framework, thereby permitting different algorithms to be easily compared.

2. We are able to create new algorithms by combining sub-algorithms in unexpected ways. In this report we describe 43 new algorithms created in this way; several of these new algorithms apparently have better performance than many previously described algorithms.
3. Because different algorithms can be easily compared, detailed qualitative performance analysis is possible. This analysis prunes the space of reasonable concurrency control algorithms to 11. In certain stereotypical situations the number of plausible algorithms can be further reduced to 3 or 4.
4. Finally, our structuring of the concurrency control problem helps organize the design of DDBMS concurrency control algorithms. This report describes the major techniques that can be employed by such algorithms, and the qualitative impact of each technique on performance.

The remainder of Section 1 is organized as follows. Section 1.2 presents examples illustrating the concurrency control problem. We will see that concurrency control is similar in appearance to synchronization problems that arise in operating systems. The relationship between these two problem areas is discussed in Section 1.3. Section 1.4 presents a section outline of the rest of the report.

1.2 Examples of Concurrency Control Anomalies

The goal of concurrency control is to prevent interference between users who are simultaneously accessing a database. In this section we illustrate the problem by presenting two "canonical" examples of inter-user interference. Both examples are stated in the context of an on-line electronic funds transfer system accessed via remote automated teller machines (ATMs). In response to customer requests, ATMs retrieve data from a database, perform computations, and store results back into the database.

Anomaly 1 -- Lost Updates

Suppose two customers simultaneously try to deposit money into the same account. In the absence of concurrency control, these two activities could interfere as shown in figure 1.1. The two ATMs handling the two customers could read the account balance at approximately the same time, compute new balances in parallel, and then store the new balances back into the database. The net effect is incorrect: although two customers deposited money, the database only reflects one activity; the other deposit has been lost by the system.

Lost Update Anamoly

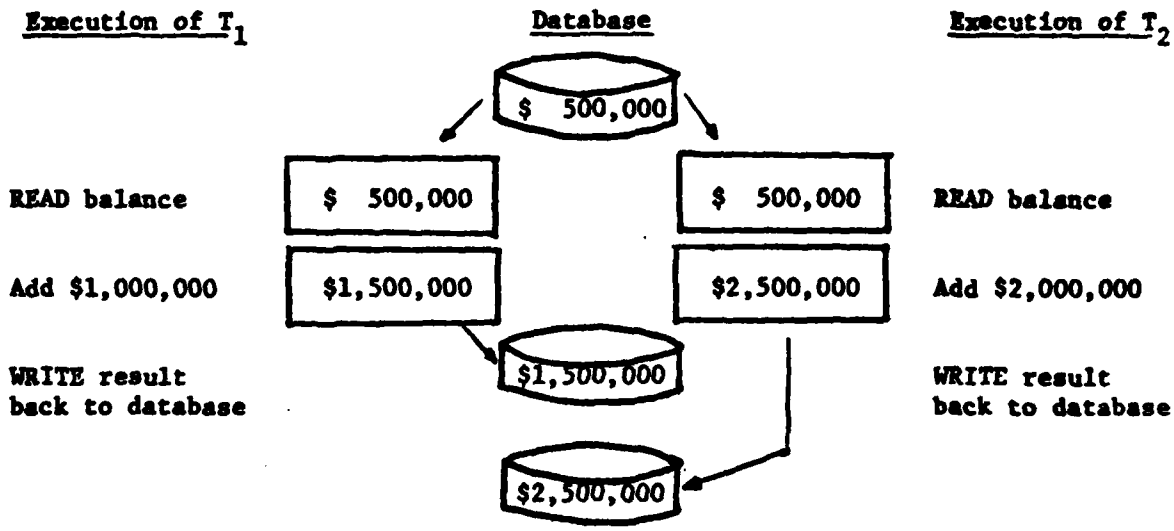
Figure 1.1

- Suppose Acme Corp.'s initial balance is $500,000. And suppose the following two transactions are simultaneously executed.

T_1: deposit $1,000,000 into Acme's account
T_2: deposit $2,000,000 into Acme's account

- The correct final balance is $3,500,000

- In the absence of concurrency control, the following incorrect execution could occur.



Anomaly 2 -- Inconsistent Retrievals

Suppose two customers simultaneously execute the following transactions.

Customer 1: Move $1,000,000 from Acme Corp.'s savings account to its checking account.

Customer 2: Print Acme Corp.'s total balance in savings and checking.

In the absence of concurrency control these two transactions could interfere as shown in figure 1.2. The first transaction might read the savings account balance, subtract $1,000,000, and store the result back in the database. Then the second transaction might read the savings and checking account balances and print the total. Then the first transaction might finish the funds transfer by reading the checking account balance, adding $1,000,000, and finally storing the result in the database.

Unlike Anomaly 1, the finally values placed into the database by this execution are correct. Nonetheless, the execution is incorrect because the balance printed by Customer 2 is $1,000,000 short.

These two examples do not exhaust all possible ways in which concurrent users can interfere, and the notion of interference-free execution will be defined precisely in later sections. However, these examples are typical of the concurrency control problems that arise in DBMSs.

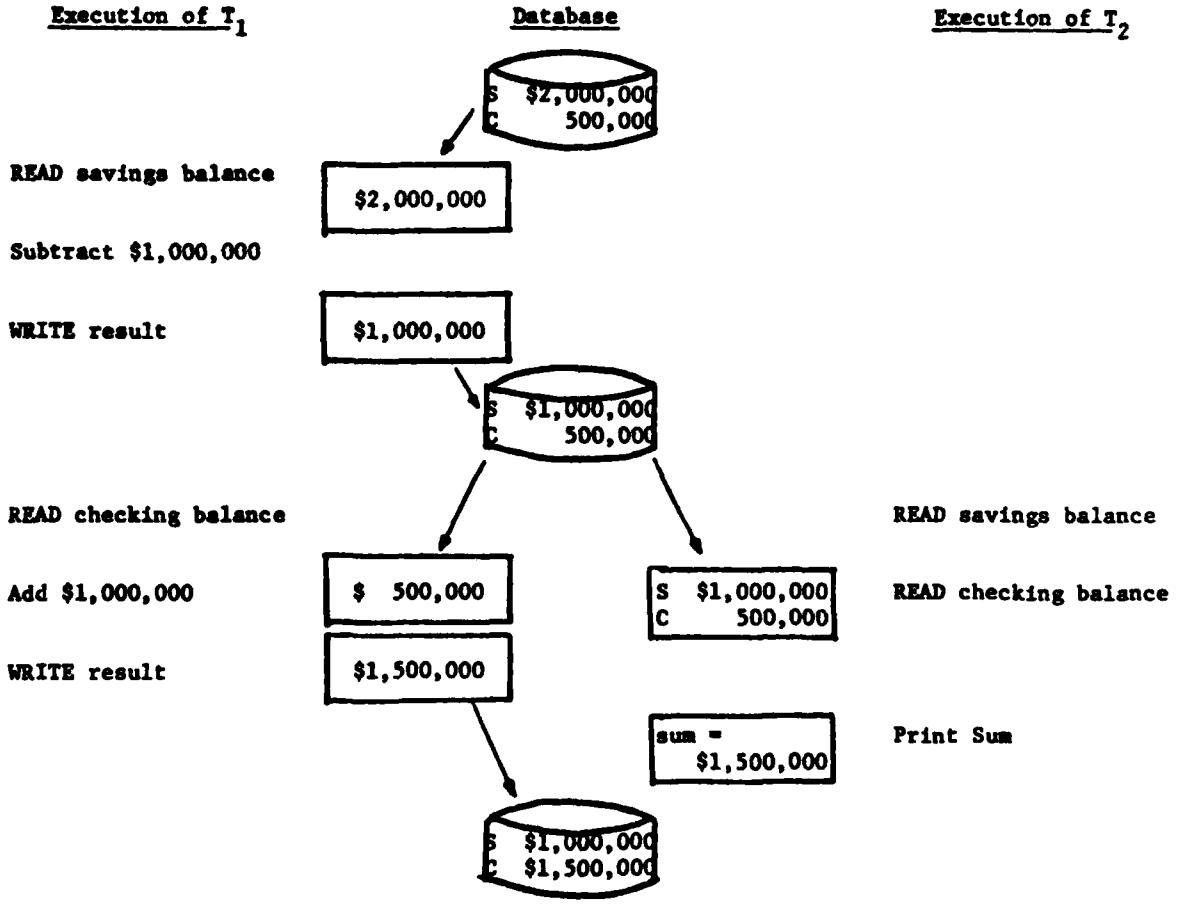
Inconsistent Retrieval Anamoly Figure 1.2

- Suppose Acme Corp. has $2,000,000 in savings and $500,000 in checking. And suppose the following two transactions are executed simultaneously.

  T_1: Move $1.000,000 from savings to checking
  T_2: Print the sum of savings and checking

- The correct final balances are $1,000,000 in savings and $1,500,000 in checking. The sum of checking and savings is $2,500,000, (both before and after T_1).

- In the absence of concurrency control, the following incorrect execution could occur.



1.3 Comparison to Mutual Exclusion Problems

The problem of database concurrency control is similar in some respects to the problem of mutual exclusion in operating systems. The latter problem is concerned with coordinating access by concurrent processes to system resources -- e.g. memory, I/O devices, CPU, etc. The basic problem is to ensure that each resource is accessed by at most one process at any given time. Many techniques for solving this problem have been developed including locks, semaphores [Dijkstra], monitors [Hoare], and serializers[Hewitt].

The concurrency control and mutual exclusion problems are similar in that both problems are concerned with controlling concurrent access to shared resources. However, there are also a number of major differences between these problems.

Most importantly, concurrency control and mutual exclusion have fundamentally different objectives. The goal of a concurrency control algorithm is to ensure that concurrent processes do not "interfere" with each other. The goal of a mutual exclusion algorithm is less ambitious, seeking only to ensure that concurrent processes do not access the same resource at the same time.

This distinction substantially impacts the technical content of each problem area, as illustrated by the following example. Suppose processes P_1 and P_2 require access to resources R_1 and R_2 at different points in their execution. In an operating system, the following interleaved execution of these processes is perfectly acceptable: P_1 uses R_1; P_2 uses R_1; P_2 uses R_2; P_1 uses R_2. In a database, however, this execution is not always acceptable. By way of proof, Anomaly 2 of the previous section results from an interleaved execution of this form, where R_1 represents Acme Corp.'s savings account and R_2 represents its checking account.

Other differences between concurrency control and mutual exclusion are discussed in [CBT].

1.4 Outline of Report

The report consists of three main parts. Part I, comprising Sections 1-3, is introductory in nature. Section 1 describes the concurrency control problem in general terms. Section 2 introduces standard terminology for defining concurrency control algorithms. Section 3 defines the concurrency control problem in precise terms and states the precise decomposition of this problem into the sub-problems of read-write and write-write synchronization.

Part II -- Sections 4-6 -- is the technical body of the report. Section 4 presents a series of techniques for solving the read-write and write-write synchronization sub-problems. Section 5 shows how these techniques can be integrated to form complete solution to the DDBMS concurrency control problem; those solutions are called concurrency control methods. Section 6 analyzes the performance of principal concurrency control methods in qualitative terms.

The final part of the report, consisting of Section 7, summarizes our results and recommends problems for further research.

In addition, there is an Appendix describing three concurrency control methods that do not fit into the framework of Sections 4-6. These methods are intellectually interesting, but appear to be too inefficient for practical use.

2. Transaction Processing Model

A concurrency control algorithm is a component of a distributed database management system (DDBMS). To understand how a concurrency control algorithm operates, one must understand how the algorithm fits into the overall DDBMS. In this section we present a simple model of a DDBMS, emphasizing how the DDBMS processes transactions. In later sections we shall describe concurrency control algorithms in terms of this model.

2.1 Preliminary Definitions

Distributed Database Management System

In this report, let us consider a distributed database management system (DDBMS) to be a collection of sites interconnected by a network. Each site is a computer running one or both of the following software modules: a transaction manager (TM) or a data manager (DM). Briefly, TMs supervise user interactions with the DDBMS while DMs manage the actual database.

A network is a computer-to-computer communication system. Sites communicate by sending messages through the network. The network is assumed to be perfectly reliable -- if site A sends a message to site B, site B is guaranteed to receive the message without error. In addition, we assume that between any pair of sites the network delivers messages in the order they were sent -- if site A sends two messages to site B, site B is guaranteed to receive the first message sent by A before the second. Network reliability and message sequencing are often implemented in software running at the sites themselves; the details of this software will not be considered in this report.

Database

Databases in practice are large, structured collections of information. For purposes of concurrency control database structure can be ignored, and a very simple database model may be adopted.*

From a user's perspective, a database consists of a collection of logical data items, denoted X,Y,Z,... We leave the

---

*In principle, database structure can be exploited to improve the performance of concurrency control algorithms. For example, [SK] describe special concurrency control algorithms for certain kinds of tree-structured data. Database structure is not exploited, however, by general-purpose concurrency control algorithms at the present state-of- the art.

granularity of logical data items unspecified. In practice, logical data items may be files, records, fields of files, or the like. In this report, the reader may think of logical data items as global variables in the style say of FORTRAN. A logical database state is an assignment of values to the logical data items comprising a database.

Each logical data item may be stored at any DM in the system or redundantly at several DMs. A stored copy of a logical data item is called a stored data item and we use $x_1,\ldots, x_m$ to denote the stored copies of logical data item X. When no confusion is possible we use the term data item in place of stored data item. A stored database state is an assignment of values to the stored data items of a database.

The impact of redurdant data has been misunderstood in most previous work on concurrency control. We will see that redundant data adds little to the complexity of the concurrency control problem, once the problem is understood in proper terms. Intuitively, there is little difference between one logical data item X with m copies $x_1,\ldots x_m$ as opposed to m logical data items $X_1,\ldots, X_m$ with one copy each. The issue of data redundancy will appear up from time to time in this report, but it is not a major source of complexity.

Transactions

Users interact with the DDBMS by executing transactions. Transactions come in many different forms. They may be on-line queries expressed in a self-contained query language (e.g. QUEL [HSW]); they may be report generating programs coded in a report writing language (e.g., RPG); or they may be application programs written in a general-purpose programming language augmented with data manipulation commands (e.g. COBOL augmented with CODASYL DML[Date]).

The concurrency control algorithms we study in this report pay no attention to the computations performed by transactions. Instead these algorithms are only concerned with the data accessed by transactions.* That is, these algorithms make all of their decisions based solely on the data items a transaction reads and the data items it writes. Consequently, the detailed form of transactions is unimportant in our analysis.

The only properties we assume of transactions are the following.

1. They represent complete and correct computations; that is, each transaction if executed alone on an initially consistent data base, would terminate, output correct results, and leave the data base consistent.

---

*In principle, concurrency control performance can be improved by exploiting the internal computations of transactions. However this requires automatic-program-understanding capabilities that are beyond the state-of-the-art.

2. They obtain data from the data base by issuing READ operations to the DDBMS and modify data by issuing WRITE operations. The arguments to these commands are logical data items, and it is the responsibility of the DDBMS to choose one stored copy of each logical data item for READs and to update all stored copies of each logical data item for WRITEs.

We model a transaction as a sequence of READ and WRITE operations, paying no attention to its internal computations.

The logical readset of a transaction is the set of logical data items the transaction reads, and the logical writeset of a transaction is the set of logical data items it writes. Stored readsets and stored writesets are defined analogously. Two transactions are said to conflict if the stored readset or writeset of one intersects the stored writeset of the other. It is a fundamental theorem of concurrency control that two transactions require synchronization only if they conflict.

Correctness of a Concurrency Control Method

The correctness of a concurrency control algorithm is defined relative to user expectations regarding transaction execution. There are two correctness criteria.

First, users expect that each transaction submitted to the system will eventually be executed. To meet this expectation

the concurrency control algorithm must avoid deadlock, indefinite postponement, and cyclic restart problems.

In addition, each user expects his transactions to execute atomically, without interference from other transactions. The computation performed by a transaction should be the same regardless of whether it executes alone in a dedicated system or in parallel with other transactions in a multi-programmed system. The attainment of this expectation is the principal issue in concurrency control. The bulk of this report is devoted to this objective.

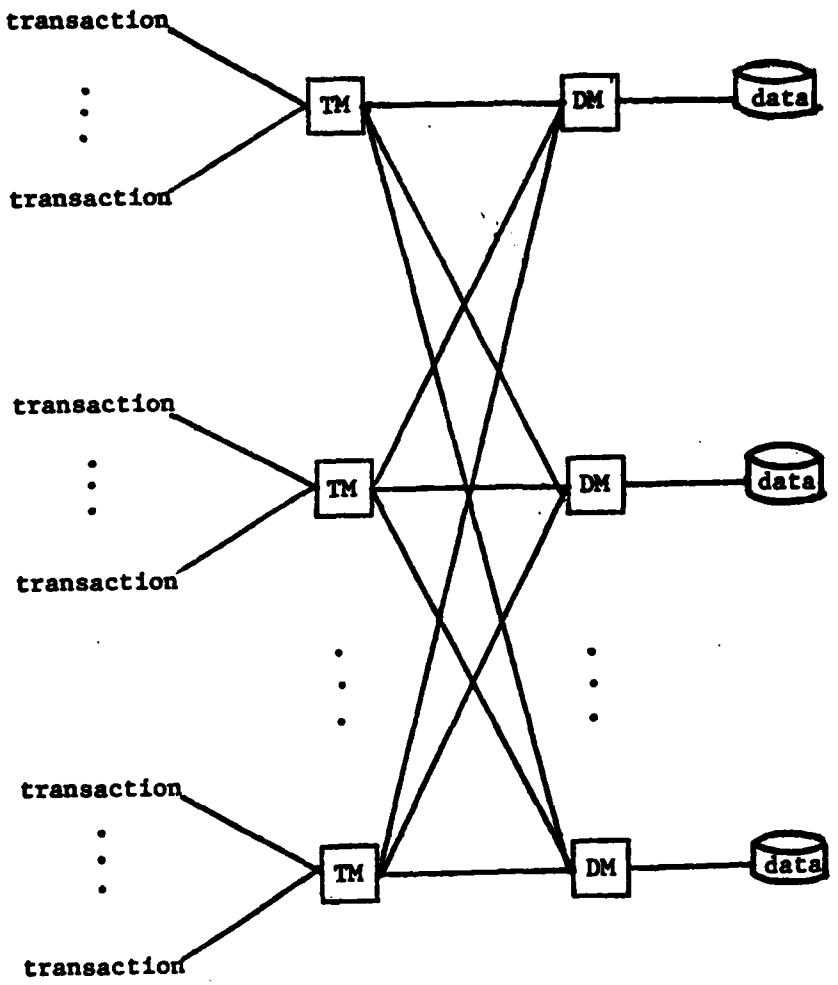
2.2 DDBMS Architecture

The architecture of our system model is illustrated in figure 2.1. There are four components: transactions, TMs, DMs, and data. Transactions communicate with TMs, TMs communicate with DMs, and DMs manage the data. Note that TMs do not communicate with other TMs, nor do DMs communicate with other DMs. (We will see minor exceptions to this strict partitioning in Section 4). The interface between transactions and TMs is called the external interface of the DDBMS; the interface between TMs and DMs is called the internal interface.

TMs are responsible for supervising transactions. Each transaction executed in the DDBMS is supervised by a single TM,

DDBMS System Architecture

Figure 2.1



meaning that the transaction issues all of its database operations to that TM. Any distributed computation that is needed to execute the transaction is managed by the TM. Therefore from the perspective of any individual transaction, the system consists of a single TM and multiple DMs. This view of the system architecture is illustrated in figure 2.2.

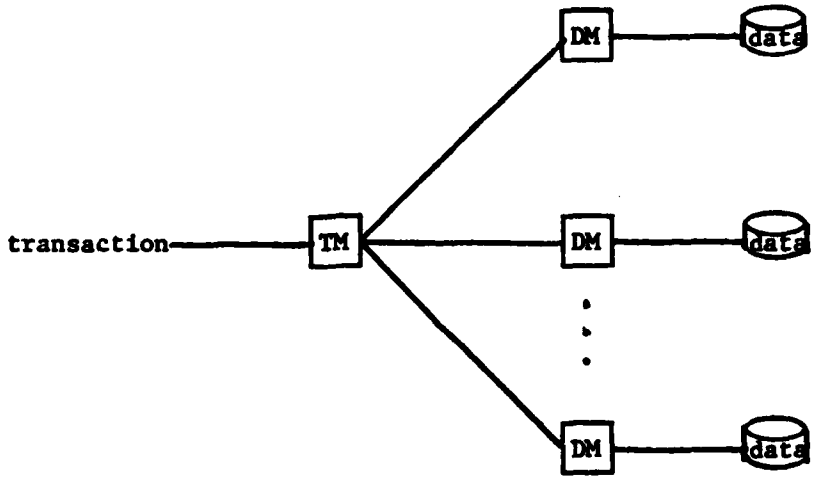
Four operations are defined at the external interface. Let X be any logical data item. READ(X) returns the value of X in the current logical database state. WRITE(X, new-value) creates a new logical database state in which X has the specified new value. In addition, since transactions are assumed to represent complete computations, we need BEGIN and END operations to bracket transaction executions. The BEGIN and END issued by a transaction tell that transaction's TM when it starts and finishes executing.

DMs are responsible for managing the stored database, functioning essentially as back-end database processors. In response to commands from transactions, TMs issue commands to DMs specifying stored data items to be read or written. The details of the TM-DM interface constitutes our core of a transaction processing model. These details are discussed in Sections 2.3 and 2.4.

In Section 2.3 we describe the TM-DM interaction in a centralized database environment. In Section 2.4 we extend the discussion to a distributed database setting.

System Architecture as Seen by
Individual Transactions

Figure 2.2



2.3 Centralized Transaction Processing Model

A centralized DBMS consists of one TM and one DM executing at the same site. Transactions access the DBMS by issuing BEGIN, READ, WRITE, and END operations, as described in Section 2.2. These operations are processed as follows.

When a transaction T issues its BEGIN operation, the TM initializes a private workspace for T. The private workspace functions as a temporary buffer for values that T wishes to write into the database, and as a cache for values that T reads from the database.

When T issues a READ(X) operation, the TM checks the private workspace to see if the workspace contains a copy of X. If the workspace contains a copy of X, the value of that copy is returned to T. Otherwise the TM issues a command to the DM requesting that the stored copy, x of X be retrieved from the database. This operation is denoted dm-read(x). The value retrieved by the DM is given to T and is also placed into the private workspace for future reference.

When T issues a WRITE(X, new-value operation), the TM again checks the private workspace. If the workspace has a copy of X, its value is updated to new-value; otherwise a copy of X with

that value is created and placed in the workspace. The new value of X is not propagated to the stored database at this time.

The only time the stored database is updated is when T issues its END operation. In response to the END operation, the TM issues an operation denoted dm-write(x) for each logical data item X updated by T. Each dm-write(X) operation requests that the DM update the value of X in the stored database; the value written into x is the value of the copy of x in T's local workspace. When all dm-writes are processed, the execution of T is finished, and the private workspace is discarded.

The DBMS is at liberty to restart T at any point before a dm-write command has been processed. The effect of restarting T is to obliterate its private workspace and to re- execute T from the beginning. As we will see, many concurrency control algorithms use transaction restarts as a tactic for attaining interference-free executions. However, once a single dm-write has been processed, T cannot be restarted. This is because each dm-write permanently installs an update into the database, and we cannot permit the database to reflect partial effects of transactions.

External vs. Internal Operations

It is important to distinguish the operations issued by transactions from those that are applied to the database. From the transactions' viewpoint, the operations that access the database are READ and WRITE. However from the system's viewpoint, most READ operations and all WRITE operations access the private workspace; indeed from the system's viewpoint, END is the operation that causes new values to be written into the database.

Therefore when we say "a write (or a read) has been processed" we must be careful to say who is doing the "writing" (or "reading"). We use READ and WRITE to refer to the transactions' operations, and dm-read and dm-write to refer to operations against the stored database. READ and WRITE are external operations; dm-read and dm-write are internal ones.

Correctness and Performance Issues

To be correct, the DBMS must only allow interference-free executions. A trivial way to attain this goal is to execute transactions serially, i.e., one at a time with no interleaving of operations from different transactions. However system resources would be poorly utilized by this approach and transactions would suffer long delays while waiting to be processed. For performance reasons, interleaving of operations from different transactions is essential.

The challenge of concurrency control is to synchronize interleaved operations so that these operations do not interfere with each other.

Reliability Issues

A database system can fail in many ways and a detailed treatment of reliability issues is beyond the scope of this report. However, one particular reliability problem has a major impact on concurrency control, namely the problem of atomic commitment.

The problem of atomic commitment is the following. Consider a transaction T that updates data items x,y,z,... and suppose the DBMS fails while processing T's END command. If this occurs, it may be the case that some of T's updates have been installed into the stored database while others have not. For example, the DBMS may have crashed after processing the dm-write for x, but before processing the dm-writes for y,z,... As a result, the database will contain incorrect information. Figure 2.3 illustrates this phenomenon.

To avoid this problem, the DBMS must have the ability to "atomically commit" all of a transaction's dm-writes. I.e., the DBMS needs a mechanism for ensuring that all of a transaction's dm-writes are processed or none are.

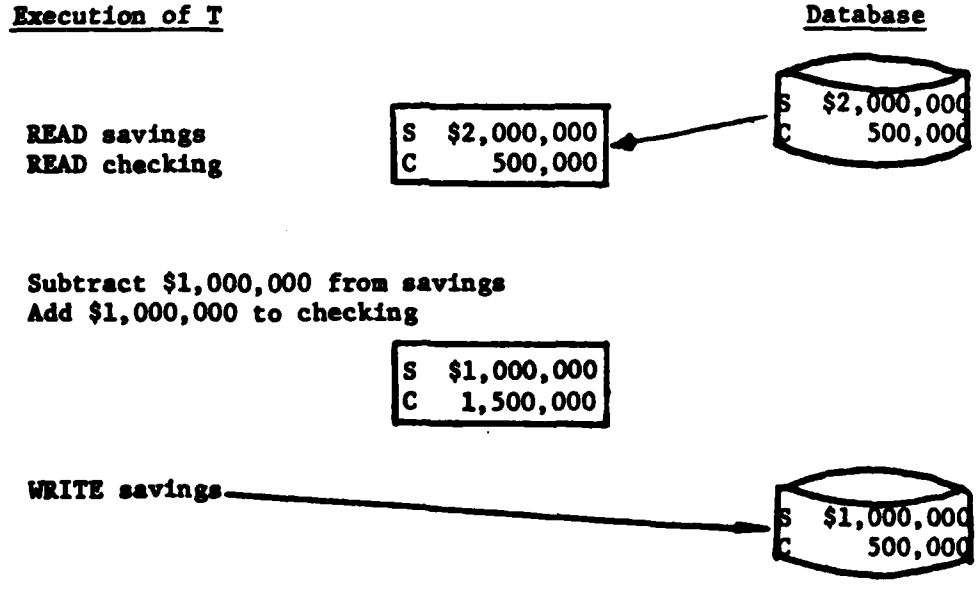
The "standard" way to implement atomic commitment involves a procedure called two-phase commit.* Again consider a transaction

The Need for Atomic Commitment
Figure 2.3

- Consider a database of banking information
- Suppose Acme Corp.'s savings account has $2,000,000 and its checking account has $500,000. And suppose the DBMS fails while processing the following transaction.

T: Move $1,000,000 from savings to checking

- In the absence of atomic commitment, the following incorrect execution could occur.



T that is updating x,y,z,... When T issues its END command, the first phase of two-phase commit begins. During this phase the DM copies the values of x,y,z,... from T's private workspace, placing these values in secure storage (typically magnetic tape). If the DBMS should happen to fail during the first phase, no harm is done, since none of T's updates have yet been applied to the stored database.

During the second phase, the DBMS copies the values of x,y,z,... into the stored database. If the DBMS fails during the second phase, the database may contain incorrect information. However since the values of x,y,z,... are stored on secure storage, this inconsistency can be rectified when the system recovers: the recovery procedure reads the values of x,y,z,... from secure storage and resumes the commitment activity.

We will see in Section 2.4 that atomic commitment is harder to implement in a distributed DBMS. As a prelude to that discussion it is convenient to introduce a third TM-DM operation; called pre-commit. This operation instructs the DM to copy a data item from the private workspace to secure

---

*The term "two-phase commit" is commonly used to denote the distributed version of this procedure, which we describe in the next section. However, since the centralized and distributed versions are essentially identical in structure, we use "two-phase commit" to describe both.

storage. (Pre-commit is short for prepare-to-atomically-commit).

Summary of Centralized Transaction Processing Model

We may summarize our model of transaction processing in a centralized DBMS as follows. Let T be a transaction.

1. When T issues its BEGIN operation, the TM creates a private workspace for T.
2. When T issues a READ(X) operation, there are two cases: (i) if the private workspace contains a copy of X, the value of that copy is given to T; else (ii) the TM issues a dm-read(X) operation to the DM which retrieves the stored value of X from the database.
3. When T issues a WRITE(X, new-value) operation, there are also two cases: (i) if the private workspace contains a copy of X, the value of that copy is changed to new-value; else (ii) a copy of X with that value is created.
4. When T issues its END operation, two-phase commit begins. For every logical data item X updated by T, the TM issues a pre-commit(X) to the DM. This causes the DM to copy the value of X from the private workspace to secure storage. After all pre-commits are processed, the TM issues a dm-write(x) operation for every X updated by T. This causes the DM to update the

value of X in the stored database to the value of X in secure storage.

After all dm-writes are processed, T is finished.

2.4 Distributed Transaction Processing Model

A distributed database system has many TMs and many DMs configured as per Section 2.2. As described in that section, each transaction executed in the system is supervised by a single TM. From the perspective of an individual transaction, therefore, the DDBMS consists of one TM and multiple DMs.

Our model of transaction processing in a distributed environment is similar in outline to the centralized case. The main differences lie in two areas: how private workspaces are handled, and the implementation of two-phase commit.

Private Workspaces in a DDBMS

In a centralized DBMS we assumed that private workspaces were part of the TM. We also assumed that data could freely move between a transaction and its workspace, and between a workspace and the DM.

These assumptions are not appropriate in a distributed environment because TMs and DMs often run at different sites.

The movement of data between a TM and a DM often entails inter-site communication which can be quite expensive. To reduce this cost, many DDBMSs employ query optimization procedures which regulate (and hopefully reduce) the flow of data between sites [ESW, GBWRR, HY Willcox, Wong].

For example, in SDD-1 the private workspace for a transaction T is not usually located at the same site as T's TM [GBWRR]. Instead, the workspace is distributed across all sites at which T accesses data. Although conceptually this workspace is controlled by T's TM, it is misleading to say that the workspace is part of the TM. The details of how T reads and writes data in these workspaces is a query optimization problem, and has no direct effect on concurrency control. Consequently, we factor this issue out of our model for distributed transaction processing.

The resulting transaction processing model is summarized below.

Summary of Distributed Transaction Processing Model

Let T be a transaction. T is processed in a DDBMS as follows.

1. When T issues its BEGIN operation, T's TM creates a private workspace for T. The location and organization of this workspace is not specified in the model.
2. When T issues a READ(X) operation, the TM checks the private workspace to see if a copy of X is present. If

so, the value of that copy is made available to T. Otherwise the TM selects some stored copy of X, say x_i, and issues a dm-read(x_i) operation to the DM at which x_i is stored. The DM responds by retrieving the stored value of x_i from the database, placing this value in the private workspace. The TM then makes this value available to T.

3. When T issues a WRITE(X, new-value) operation, the value of X in the private workspace is updated to new-value, assuming the workspace contains a copy of X. Otherwise, a copy of X with the new value is created in the workspace.

4. When T issues its END operation, two-phase commit begins. For each logical data item X updated by T, and for each stored copy x_i of X, the TM issues a pre-commit(x_i) operation to the DM that stores x_i. The DM responds by copying the value of X from T's private workspace, placing this value onto secure storage internal to the DM. After all pre-commits are processed, the TM issues dm-write operations for all copies of all logical data items updated by T. A DM responds to dm-write(x_i) by copying the value of x_i from secure storage, into the stored database.

When all dm-writes have been processed, the execution of T is complete.

This transaction processing model is more abstract than the centralized model, and it leaves many details unspecified. For example, we do not specify where the private workspace is located; we also do not specify how the TM checks the workspace to see if a copy of X is present; nor do we specify how data moves between T and its workspace, and between DMs and the workspace.

These details are important to overall DDBMS functioning, but are not important insofar as concurrency control is concerned. We ignore these issues because they render our transaction processing model both simpler and more general.

This transaction processing model is used in the remainder of this report as a framework for describing and analyzing concurrency control algorithms.

Two-Phase Commit in a DDBMS

The problem of atomic commitment is aggravated in a DDBMS by the possibility of one site failing while the remainder of the system continues to operate.

Consider a transaction T that is updating data items x,y,z,... stored at DM_x, DM_y, DM_z,... respectively. Suppose T's TM fails

during the second phase of two-phase commit. For example, suppose the TM fails after issuing the dm-write(x) operation, but before issuing dm-write(y), dm-write(z),... At this point, the database contains incorrect information: the database stored at DM_x reflects the execution of T, but the databases at DM_y, DM_z,... do not. In a centralized DBMS, this phenomenon is not harmful because no transaction can access the database until the TM recovers from the failure. However, in a distributed DBMS, other TMs remain operational, and the incorrect database can be accessed from these TMs.

To avoid this problem, each DM that receives a pre-commit must be able to determine which other DMs are involved in the commitment activity. (This information could be included as a parameter to the pre-commit operation, or it could be stored in a private workspace, etc.) If T's TM fails before issuing all dm-writes, the DMs whose dm-writes were not issued can recognize the situation. These DMs then consult all DMs involved in the commitment to determine whether <u>any</u> DM received a dm-write. If <u>any</u> DM received a dm-write, the remaining ones act as if they had also received the command. Thus, if any DM applies an update to the database, they all do.

This commitment algorithm is described in greater detail in [HS2].

3. Concurrency Control Theory

In this section we review the mathematical theory of concurrency control with two objectives: to define the correctness of a concurrency control method in precise terms; to decompose the concurrency control problem into more tractable sub-problems. In Section 3.1 we present a mathematical definition of interference - free executions and in Section 3.2 we present a concise characterization of these executions. Section 3.3 uses this characterization to decompose the concurrency control problem into two major sub-problems - read-write synchronization and write-write synchronization. This decomposition is the cornerstone of our approach to the analysis and design of distributed concurrency control algorithms.

3.1 Serializability

The intuitive notion of an interference-free execution is modelled mathematically by the concept of serializability. Serializability is the formal notion of correctnes in database concurrency control.

Consider a collection of transactions, $T_1, \ldots, T_n$, and let E denote an execution of these transactions. E is called a serial execution if no transactions ever execute concurrently in E -- i.e., each transaction is executed to completion before the next one begins. Every serial execution is defined to be correct. To justify this notion of correctness, we observe that the properties of correct transactions (see Section 2.1) imply that a serial execution terminates properly and preserves database consistency. An execution is serializable if it is computationally equivalent to a serial execution. That is, a serializable execution produces the same output and has the same effect on the database as some serial execution. Since serial executions are correct and every serializable execution is equivalent to a serial one, every serializable execution is also correct. The goal of database concurrency control is to allow only serializable executions to occur.

3.2 Characterizing Serializability

The only operations that access the stored database are dm-read and dm-write. Only the relative order of dm-reads and dm-writes on behalf of different transactions can affect the computation performed by those transactions. So, insofar as serializability is concerned, it is sufficient to model an execution of transactions by the execution of dm-read and dm-write operations at the various DMs of the DDBMS.

In this spirit, we formally model an execution of transactions by a set of logs, one log per DM. Each log indicates the order in which dm-reads and dm-writes are processed at one DM. Figure 3.1 illustrates several logs. Mathematically, we define a log as a string of dm-read and dm-write operations. Modelling an execution as a set of logs is the first step in understanding serializability.

According to Section 3.1, an execution is serializable if it is computationally equivalent to a serial execution of the same transactions. To model this statement using logs, we must state the conditions under which a set of logs models a serial execution and the conditions under which two sets of logs are computationally equivalent.

3.2.1 Serial Executions

An execution modelled by a set of logs is serial if

1. for each log, and for each pair of transactions T_i and T_j whose operations appear in the log, either all of T_i's operations precede all of T_j's operations, or vice versa; and
2. for each pair of transactions, T_i and T_j, if T_i's operations precede T_j's operations in one log, then

Modelling Executions as Logs Figure 3.1

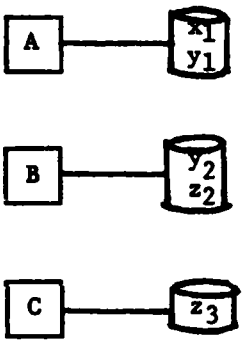
Transactions

T_1: BEGIN;
READ(X); WRITE(Y); END

T_2: BEGIN;
READ(Y); WRITE(Z); END

T_3: BEGIN;
READ(Z); WRITE(X); END

Database



- One possible execution of T_1, T_2, and T_3 is represented by the following logs. (Note: $r_i[x]$ denotes the operation dm-read(x) issued by T_i; $w_i[x]$ has the analogous meaning)

Log for DM A: $r_1[x_1]\ w_1[y_1]\ r_2[y_1]\ w_3[x_1]$

Log for DM B: $w_1[y_2]\ w_2[z_2]$

Log for DM C: $w_2[z_3]\ r_3[z_3]$

T_i's operations precede T_j's operations in every log in which operations from both T_i and T_j appear.

These conditions are illustrated in figure 3.2. Intuitively, condition (1) says that at each DM no two transactions are interleaved. Condition (2) says that transactions execute in the same order at all DMs. More precisely, condition (2) says there is a total ordering of transactions, that is consistent with the non-interleaved (i.e., serial) ordering at each DM.

3.2.2 Equivalent Executions

Two operations conflict if they operate on the same data item and one of the operations is a dm-write. The order in which operations execute is computationally significant if and only if the operations conflict. That is, if two operations conflict the order in which they execute will (in general) affect the computation they perform, while if two operations do not conflict, the order in which they execute never affects the computation.

To illustrate the notion of conflict, consider a data item x and transactions T_i and T_j. If T_i issues a dm-read(x) and T_j issues a dm-write(x), the value read by T_i will (in general) be different depending on whether the dm-read precedes or follows

Serial and Non-Serial Logs Figure 3.2

- The execution modelled in figure 3.1 is serial. Condition (1) holds since each log is itself serial--i.e., there is no interleaving of operations from different transactions. Condition (2) holds since at DM A, T_1 precedes T_2 precedes T_3; at DM B, T_1 precedes T_2; and at DM C, T_2 precedes T_3.

- The following execution is not serial; it satisfies (1) but not (2).

DM A: $r_1[x_1]$ $w_1[y_1]$ $r_2[y1]$ $w_3[x_1]$

DM B: $w_2[z_2]$ $w_1[y_2]$

DM C: $w_2[z_3]$ $r_3[z_3]$

- The following execution is also not serial; it doesn't satisfy (1) or (2).

DM A: $r_1[x_1]$ $r_2[y1]$ $w_3[x_1]$ $w_1[y_1]$

DM B: $w_2[z_2]$ $w_1[y_2]$

DM C: $w_2[z_3]$ $r_3[z_3]$

the dm-write. Similarly, if both transactions issue dm-write(x) operations, the final value of x depends on which dm-write happens last. Those conflict situations are called read-write conflicts and write-write conflicts respectively.

On the other hand, if T_i and T_j both issue dm-read(x) operations, the value read is not affected by the order in which these dm-reads are executed. Similarly, if T_i reads x and T_j writes into a different data item y, or if T_i writes x and T_j writes y, the order of execution does not affect the results. These are, of course, non-conflict situations.

The notion of conflict can be used to characterize the equivalence of executions.

Let E_1 and E_2 be two executions, modelled by the logs $\{L_{1,1},\ldots, L_{1,n}\}$ and $\{L_{2,1},\ldots L_{2,n}\}$ respectively, where $L_{i,j}$ models the execution at DM_j for E_i. E_1 and E_2 are computationally equivalent if the following condition holds [PBR, Papadimitriou]: for each j, $1<j<n$, $L_{1,j}$ and $L_{2,j}$ contain the same set of dm-reads and dm-writes and each pair of conflicting operations appears in the same relative order in both logs.

Intuitively, computational equivalence must hold in this case for two reasons:

1. each dm-read operation reads data item values that were produced by the same dm-writes in both executions; and
2. the final dm-write on each data item is the same in both executions.

The first condition ensures that each transaction reads the same input in both executions (and therefore performs the same computation). Combining this with the second condition, we can conclude that both executions leave the database in the same final state.

3.2.3 Serializable Executions

Using our formal concepts of serial execution and computational equivalence, we can now characterize serializable executions. The following theorem states the characterization precisely.

Theorem 1 [PBR, Papadimitriou, SLR] Let $\underline{T}=\{T_1,\ldots, T_n\}$ be a set of transactions and let E be an execution of these transactions modelled by the set of logs $\{L_1,\ldots, L_n\}$. E is serializable if there exists a total ordering of $\underline{T}$ such that for each pair of conflicting operations O_i and O_j from distinct transactions T_i and T_j (resp.), O_i precedes O_j in a log iff T_i precedes T_j in the total ordering.

The total order hypothesized in Theorem 1 is called a serialization order. A serialization order indicates a serial execution of the transactions T that is computationally equivalent to the original execution E. That is, if the transactions had executed serially in the hypothesized order, then the computation performed by the transactions would have been identical to the computation represented by E. The main step in proving the theorem is to observe that the theorem requires that conflicting operations appear in the same order in E as in the hypothetical serial execution; the equivalence of E and the serial execution follows from Section 3.2.2.

To attain serializability, the DDBMS must guarantee that all executions satisfy the condition of Theorem 1. Those conditions require that conflicting dm-reads and dm-writes be processed in certain relative orders. Concurrency control is the activity of controlling the relative order of conflicting operations; an algorithm to perform such control is called a synchronization technique. So, to be correct, a DBMS must incorporate synchronization techniques that guarantee the conditions of Theorem 1.

3.3 A Paradigm for Concurrency Control

In Theorem 1, read-write conflicts and write-write conflicts are treated together under the general notion of conflict. Viewed together, they contribute equally to the serializability of an execution. However, we can decompose the concept of serializability by distinguishing these two types of conflict. To do so, we first introduce additional notation.

3.3.1 The -> Relation

Let E be an execution modelled by a set of logs. We define three binary relations on transactions in E, denoted ->rw, ->wr, and ->ww. For each pair of transactions, T_i and T_j

1. T_i ->rwT_j iff in some log of execution E, T_i reads some data item into which T_j subsequently writes;
2. T_i ->wr T_j iff in some log of execution E, T_i writes into some data item that T_j subsequently reads;
3. T_i ->wwT_j iff in some log of execution E, T_i writes into some data item into which T_j subsequently writes.

We denote the union of the first two relations by ->rwr, i.e., ->rwr = (->rw U ->wr). The union of all three relations is denoted ->.

Intuitively, we can interpret -> (with any subscript) to mean "in any serialization must precede". For example, T_i ->rw T_j means "T_i in any serialization must precede T_j". From Theorem 1, we can see why this interpretation is accurate. If T_i reads some data item x before T_j writes into x, then the hypothetical serialization in Theorem 1 must have T_i preceding T_j.

Every conflict between operations in E is represented by an -> relationship. Therefore, we can restate Theorem 1 in terms of ->. According to Theorem 1, E is serializable if there is a total order of transactions that is consistent with the order of all conflicts. In terms of ->, this means that E is serializable if there is a total order of transactions that is consistent with all -> relationships. This latter condition holds iff -> is acyclic. (A relation, ->, is acyclic if there is no sequence i_1 -> i_2, i_2 -> i_3,..., i_n such that $i_1 = i_n$.) We state this conclusion as Theorem 1' [BSW].

Theorem 1' E is serializable if it has an acyclic -> relation.

3.3.2 Distinguishing Read-Write from Write-Write Synchronization

The advantage of Theorem 1' over Theorem 1 is that we can decompose -> into its components, ->rwr and ->ww, and restate the theorem in terms of these components.

Theorem 2 Let ->rwr and ->ww be associated with execution E. Then E is serializable if (a) ->rwr and ->ww are acyclic, and (b) there is a total ordering of the transactions that is consistent both with all ->rwr relationships and all ->ww relationships.

Theorem 2 is equivalent to Theorems 1 and 1', but it emphasizes a point that is overlooked in the earlier theorems: read-write and write-write conflicts interact only insofar as there must be a total ordering of the transactions consistent with both types of conflicts. This suggests that read-write and write-write conflicts can, to some extent, be synchronized independently. Herein lies the significance of Theorem 2.

Theorem 2 tells us that we can use different techniques for synchronizing read-write and write-write conflicts within a single system. One technique can be used to guarantee an

acyclic ->rwr relation (which amounts to read-write synchronization) while a different technique is used to guarantee an acyclic ->ww relation (write-write synchronization). However, Theorem 2 says that having both ->rwr and ->ww acyclic is not enough. There must also be one serial order that is consistent with all -> relations. This serial order is the cement that binds together the read-write and write- write synchronization techniques.

The decomposition of the serializability problem into the problems of synchronizing read-write and write-write conflicts is the cornerstone of our paradigm for understanding distributed DBMS concurrency control.

This paradigm tells us that every correct concurrency control algorithm can be analyzed as a composition of a read-write and a write-write synchronization technique. The paradigm focuses attention on the "essence" of each synchronization technique -- namely the way in which the technique guarantees an acyclic ->rwr or ->ww relation. The paradigm permits us to analyze each technique in the abstract, outside the context of any complete concurrency control method. And the paradigm guides us in the design of new concurrency control methods, by explaining how read-write and write-write synchronization techniques must be "glued" together to form a correct method.

4. Synchronization Techniques

This section begins the technical body of the report. In this section and the next we describe a large number of concurrency control algorithms.

In section 6 we examine the performance of those algorithms.

Our presentation is structured by the paradigm of Section 3.3. We decompose the overall concurrency control problem into two major sub-problems: rw synchronization and ww synchronization.* In Section 4 we describe algorithms that accomplish rw and/or ww synchronization, then in Section 5 we show how to combine rw and ww synchronization algorithms into a correct control algorithm.

It will be important in the remainder of this report to distinguish algorithms that attain rw and/or ww synchronization from algorithms that solve the entire concurrency control problem in a distributed database environment. We shall use the term synchronization technique for the former type of algorithm, and concurrency control method for the latter.

---

*Hereafter we use "rw" and "ww" as abbreviations for "read-write" and "write-write" respectively.

All practical concurrency control methods can be analyzed as combinations and variations of two basic synchronization techniques: two phase locking (2PL) and timestamp ordering (T/O). (Two-phase locking should not be confused wth two-phase commit.) 2PL is studied in Section 4.1, and T/O is studied in Section 4.2. In each section we present an implementation-independent specification of the synchronization techniques, followed by an implementation that we deem to be the basic implementation of the technique, followed by implementation alternatives. We have attempted to list the major implementation alternatives for each technique, but do not claim to have exhausted all possible variations. In addition, we consider ancillary problems that must be solved to make each implementation effective.

4.1 Two Phase Locking (2PL)

4.1.1 Specification

Two phase locking synchronizes read and write operations by explicitly detecting and preventing conflicts between concurrent operations. Before reading a data item x, a transaction must "own" a read-lock on x. Before a transaction may write into x, it must "own" a write-lock on x. The ownership of locks is governed by two rules:

1. Different transactions cannot simultaneously own locks that conflict (defined below): and
2. Once a transaction surrenders ownership of a lock, it may never obtain additional locks.

The definition of conflicting lock depends upon the type of synchronization being performed. For rw synchronization two locks conflict iff (a) both are locks on the same data item x, and (b) one is a read-lock and the other is a write lock. For ww synchronization two locks conflict iff (a) both are locks on the same data item, and (b) both are write-locks.

The second lock ownership rule causes every transaction to obtain locks in a two phase manner. During the first phase, called the growing phase, the transaction obtains more and more locks without releasing any locks. By releasing a lock, the transaction enters the second phase, called the shrinking phase. During the shrinking phase the transaction releases more and more locks, and by rule 2, is prohibited from obtaining additional locks. When the transaction terminates (or aborts) all remaining locks are automatically released.

Several authors have proven that 2PL is a correct synchronization technique, meaning that 2PL attains an acyclic ->rwr (resp. ->ww) relation when used for rw (resp. ww) synchronization [EGLT, BSW, Papadimitriou]. The serialization order attained by 2PL is determined by the order in which transactions obtain locks. At the end of the growing phase, a transaction owns all locks that it ever will own. This point in time is called the locked point of the transaction [BSW]. Let E be an execution in which 2PL is used for rw (resp. ww) synchronization. We can prove that the ->rwr (resp. ->ww) relation induced by E is identical to the relation induced by a serial execution E' in which all transactions execute at their locked points. Thus, the locked points of E determine a serialization order for E.

4.1.2 Basic Implementation

An implementation of 2PL amounts to building a 2PL scheduler, a software module that receives lock-request and lock-release operations and processes these operations in accordance with the 2PL specification.

The basic way to implement 2PL in a distributed database is to distribute the schedulers along with the database, i.e., the scheduler for data item x is located at the DM where x is stored. In this implementation read-locks may be implicitly requested by dm-read operations and write-locks may be implicitly requested by pre-commit operations. If the requested lock cannot be granted, the operation is placed on a waiting queue for the desired data item. (This introduces the possibility of deadlock; see Sections 4.1.6-4.1.8) Write-locks are implicitly released by dm-write operations. However, to release read-locks special lock-release operations are required. These lock-release operations may be transmitted in parallel with the dm-write operations, since the dm-writes signal the start of the shrinking phase. When a lock is released, the operations on the waiting queue for that data item are processed in FIFO order. In some cases it may be preferable to process

waiting operations in non-FIFO order; this issue is discussed in Section 4.1.10.

Notice that this implementation of 2PL pays no special attention to redundant data. Suppose a logical data item X is stored redundantly at m sites and let $x_1,...,x_m$ be the stored copies of X. If basic 2PL is used for rw synchronization, a transaction may read any copy of X and need only obtain a read-lock on the copy of X it actually reads. However, if a transaction updates X, it must obtain write-locks on all copies of X, since the transaction must update all copies of X. This latter observation holds whether basic 2PL is used for rw or ww synchronization.

An important advantage of the basic implementation is that little "extra" communication between TMs and DMs is required to synchronize transactions. The only extra communication needed by this technique are the operations that release read-locks.

4.1.3 Primary Copy 2PL

Primary copy 2PL differs from the preceding technique in that it pays attention to data redundancy [Stonebracker]. One copy of each logical data item is designated the primary copy of the data item; before accessing any copy of the logical data item, the appropriate lock must be obtained on the primary copy.

In the case of read-locks, this technique requires more communication than basic 2PL. Suppose x_1 is the primary copy of logical data item X, and suppose transaction T wishes to read some other copy of X, e.g. x_i. To read x_i under primary copy 2PL, transaction T must communicate with <u>two</u> DMs -- the DM where x_1 is stored (so T can lock x_1) and the DM where x_i is stored. By contrast, under basic 2PL T would only communicate with x_i's DM. In the case of write-locks, however, primary copy 2PL does not incur extra communication. Suppose transaction T wishes to update X. Under basic 2PL, T would issue pre-commits to all copies of X -- thereby requesting write-locks on these data items -- and then would issue dm-writes to all copies. Under primary copy 2PL the same operation sequence would be required; the only difference is that under primary copy 2PL the pre-commits to $x_2 \ldots, x_m$ do not serve to request write-locks on these data items.

This suggests that primary copy 2PL may be better than basic 2PL for ww synchronization, but not for rw synchronization. As we shall see in Section 5, there is no difficulty in using basic 2PL for rw synchronization, while using primary copy for ww synchronization.

4.1.4 Voting 2PL

Voting 2PL (also called majority consensus 2PL) is another 2PL implementation that exploits data redundancy. Voting 2PL is derived from the majority consensus technique of [Thomas 1,2], and is only suitable for ww synchronization.

To understand the voting protocol, we need to examine it in the context of two-phase commit. Suppose transaction T wants to write into X. Its TM sends pre-commit operations to each DM holding a copy of X. For the voting protocol, the DM always responds immediately. It acknowledges receipt of the pre-commit and says "lock set" or "lock blocked". (In the standard locking protocol, it would not acknowledge at all until the lock is set.) After the TM receives acknowledgements from the DMs, it counts the number of "lock set" responses. If a majority of the DMs responded "lock set", then the TM behaves as if all locks were set; i.e., it enters the second commit phase and issues dm-writes. Otherwise, it waits for additional "lock set" operations from DMs that originally said "lock blocked". Deadlocks aside (see Section 4.1.8), it will eventually receive enough "lock set" operations to proceed.

Since only one transaction can hold a majority of locks on X at a time, only one transaction writing into X can be engaged in its second commit phase at any time. So, all copies of X will have the same sequence of writes applied to them. Thinking in terms of locked points, we see that a transaction's locked point occurs when it has obtained a majority of its write locks on each data item in its writeset.

When many data items are being updated by a transaction, the transaction must obtain a majority of locks on each and every data item before it commits any of its writes.

4.1.5 *Centralized 2PL*

Instead of distributing the 2PL schedulers with the database, it is possible to centralize the scheduler at a single site [AD, G-M2]. Before accessing data at any site, appropriate locks must be obtained from the centralized 2PL scheduler. So, for example, to perform dm-read(x) where x is not stored at the central site, the TM must first request a read-lock on x from the central site; after the central site acknowledges to the TM that the lock has been set, the TM can send dm-read(x) to the DM that holds x. (It is possible to save some communication by having the TM send the lock request and dm-read(x) to the central site; the central site directly forwards the dm-read(x)

to x's DM instead of acknowledging the setting of the lock to the TM; the DM then responds to the TM when the dm-read(x) has been processed.) Like primary copy 2PL, this approach tends to require more communication than basic 2PL, because dm-reads and dm-writes usually cannot implicitly request locks.

The advantages and disadvantages of centralized 2PL are discussed further in Section 6.

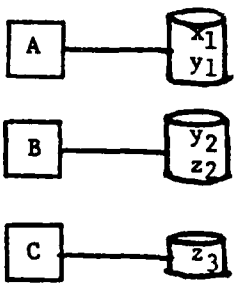
4.1.6 Deadlock

The preceding implementations of 2PL force transactions to wait for unavailable locks. If this waiting activity is uncontrolled, the possibility of deadlock exists; see figure 4.1. In this section we define a mathematical construct called a waits-for graph that is useful in characterizing deadlock situations [KC]. In Section 4.1.7-4.1.9 we use this construct to describe techniques for handling deadlocks.

A waits-for graph is a directed graph that indicates which transactions are waiting for which other transactions. The nodes of the graph represent transactions. The edges represent the "waiting-for" relationship. An edge is drawn from transaction T_i to transaction T_j if T_i is waiting for a lock currently owned by T_j. There is a deadlock in the system if and

Deadlock

Figure 4.1



- Suppose transactions execute concurrently, with each transactions issuing its READ before any transaction issues its END.
- This partial execution could be represented by the following logs.

DM A: $r_1[x_1]$

DM B: $r_2[y_2]$

DM C: $r_3[z_3]$

- At this point, T_1 has read-lock on x_1
  T_2 has read-lock on y_2
  T_3 has read-lock on z_3

- Before proceeding, all transactions must obtain write-locks.

T_1 requires write-locks on y_1 and y_2
T_2 requires write-locks on z_2 and z_3
T_3 requires write-lock on x_1

- But,

T_1 cannot get write-lock on y_2, until T_2 releases read-lock
T_2 cannot get write-lock on z_3, until T_3 releases read-lock
T_3 cannot get write lock on x_1, until T_1 releases read-lock

$\therefore$ Deadlock!

only if the waits-for graph contains a cycle, i.e., a path from some node back to itself; see figure 4.2.

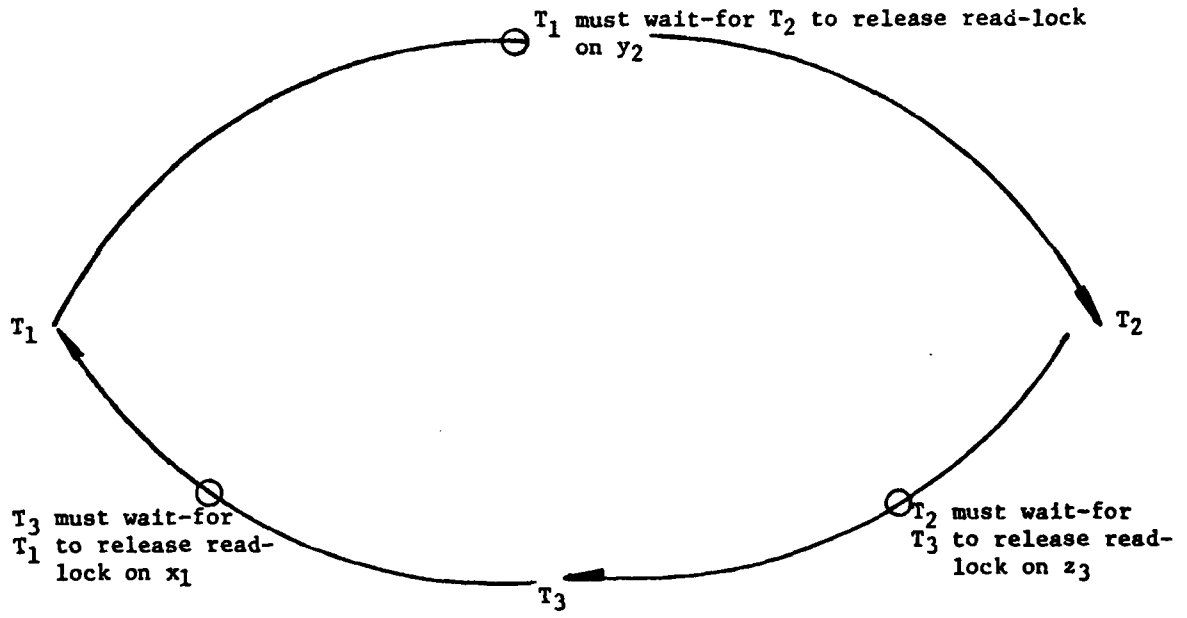
Two general techniques are available for deadlock resolution: deadlock prevention and deadlock detection. These techniques are described in the following sections.

4.1.7 Deadlock Prevention

Deadlock prevention is a "cautious" deadlock resolution scheme in which a transaction is restarted when the system is "afraid" that a deadlock might occur. To implement deadlock prevention, the 2PL scheduler described in Section 4.1.2 is modified as follows. If a lock request is denied, then the scheduler applies a "test" to the requesting transaction (say T_i) and the transaction that currently owns the lock (say T_j). If T_i and T_j pass the test, T_i is permitted to wait for T_j as usual. Otherwise, one or the other transaction is aborted. If T_i is restarted, the deadlock prevention algorithm is called non-preemptive; if T_j is restarted, the algorithm is called preemptive.

The test applied by the scheduler must guarantee that if T_i is forced to wait for T_j, then deadlock cannot result. I.e., we must ensure that the addition of edge $\langle T_i, T_j \rangle$ to the waits-for graph cannot introduce a cycle into the graph.

Waits-for Graph for Figure 4.1 Figure 4.2



Many tests have this property. One simple approach is to never let T_i wait-for T_j -- i.e., the test always outputs "false". This technique trivially prevents deadlock, but forces many restarts.

A better approach is to assign priorities to transactions and to test priorities to decide whether T_i can wait for T_j. For example, we could let T_i wait for T_j iff T_i has lower priority than T_j, (importantly, if T_i and T_j have equal priorities, T_i is not permitted to wait-for T_j or vice versa). This test is sufficient to prevent deadlock for the following reason. Imagine that we construct the waits-for graph of the system at any point in time. Every edge in the graph is guaranteed to be in priority order -- i.e., for all edges $<T_i, T_j,>$ T_i has lower priority than T_j. Since a cycle is a path from a node to itself -- i.e., a path of the form $<T_i, T_j, \ldots, T_k, T_i>$ -- and since it is impossible for T_i to have lower priority than itself, no cycle can exist in the waits-for graph. Q.E.D.

One problem with the preceding approach is that cyclic restart is possible, meaning that some unfortunate transaction could be restarted over and over again without ever finishing. To avoid this problem, [RSL] propose a technique in which "timestamps" are used to assign priorities. Intuitively, the timestamp of a transaction corresponds to the time at which it begins executing, and old transactions (ones that have been executing a long time) have higher priority than young transactions.

The technique of [RSL] requires that each transaction in the distributed system be assigned a unique timestamp. This requirement is achieved as follows [Thomas 1,2]. Timestamps are assigned to transactions by their TMs. When a transaction begins, its TM reads the local clock time and appends a unique TM identifier to the low order bits. The resulting number is the desired timestamp. The TM also agrees not to assign another timestamp until the next clock tick. This technique ensures that timestamps assigned by different TMs differ in their low order bits (because different TMs have different identifiers) while timestamps assigned by the same TM differ in their high order bits (because the TM does not use the same clock time for two different timestamps). Thus, the timestamps generated by this algorithm are unique system-wide. Notice that the correctness of this algorithm does not require that clocks at different sites be precisely synchronized.

Two timestamp-based deadlock prevention schemes are proposed by [RSL]. One, called the Wait-Die System, is a non-preemptive technique. Suppose transaction T_i tries to wait-for T_j. If T_i has lower priority than T_j -- i.e., T_i is younger than T_j -- then T_i is permitted to wait. Otherwise T_i is aborted, i.e., it "dies", and is forced to restart. It is important that T_i not be assigned a new timestamp when it restarts. The other scheme, called Wound-Wait is the preemptive counterpart to Wait-Die; if T_i has higher priority than T_j, then T_i waits, otherwise T_j is aborted.

Both Wait-Die and Wound-Wait avoid cyclic restart, but they behave quite differently: in Wound-Wait an old transaction may be restarted many times, while in Wait-Die old transactions never restart. It is suggested in [RSL] that Wound-Wait induces fewer restarts in total, but the justification is more intuitive than analytic.

Care must be exercised in using preemptive deadlock prevention schemes in conjunction with two-phase commit. For two-phase commit to operate properly, a transaction must not be aborted once the second phase of two-phase commit has begun. Notice that if the second phase of commitment has begun, the transaction in question is guaranteed not to be waiting for any other transactions, and so is guaranteed not to be involved in any deadlocks. Therefore, if a preemptive technique wishes to abort T_j, it first checks with T_j's TM to determine whether T_j has entered the second phase of commitment. If T_j has entered the second phase, no deadlock is possible and T_j is permitted to complete; otherwise T_j is aborted.

4.1.8 Deadlock Detection

Deadlock detection is an alternative to deadlock prevention. The idea is to let transactions wait-for each other in an uncontrolled manner and only abort transactions if a deadlock actually occurs. Deadlocks are detected by explicitly constructing the waits-for graph of the system and searching for cycles in that graph. (Cycles in a graph can be found efficiently using, for example, Alg. 5.2 in [AHU]). If a cycle is found, one of the transactions on the cycle is aborted, thereby breaking the deadlock. The transaction that is aborted is called the victim. To minimize the cost of restarting the victim, victim selection is usually based on the amount of resources used by each transaction on the cycle.

The principal difficulty in implementing deadlock detection in a distributed database is constructing the waits-for graph efficiently. Each 2PL scheduler can easily construct the waits-for graph based on the waits- for relationships local to that scheduler. However, these local waits-for graphs are not sufficient to characterize all deadlocks in the distributed system. Figure 4.3 illustrates such a case. Instead, it is necessary to "combine" the local waits-for graphs into a more

Multi-site Deadlock
Figure 4.3

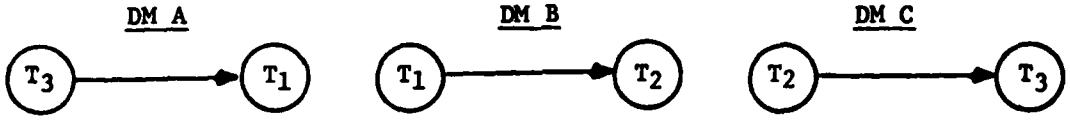
- Consider the execution illustrated in figures 4.1 and 4.2

- Locks are requested at DMs in the following order

DM A	DM B	DM C
read-lock x_1 for T_1	read-lock y_2 for T_2	read-lock z_3 for T_3
write-lock y_1 for T_1	write-lock z_2 for T_2	
*write-lock x_1 for T_3	*write-lock y_2 for T_1	*write-lock z_3 for T_2

- None of the *'ed locks can be granted and the system is in deadlock.

  However, the waits-for graphs at each DM are acyclic.



"global" waits-for graph. (Notice that centralized 2PL does not have this problem, since there is only one scheduler in that case.)

We shall describe two techniques for constructing global waits-for graphs: centralized deadlock detection and hierarchical deadlock detection.

Centralized Deadlock Detection

In the centralized approach, one site is designated the deadlock detector for the distributed system [Gray, Stonebraker]. Periodically -- e.g., every few minutes -- each scheduler transmits its local waits-for graph to the deadlock detector.* The deadlock detector combines the local graphs into a system-wide waits-for graph by constructing the union of the local graphs.

Hierarchical Deadlock Detection

An alternative approach is hierarchical deadlock detection [MM]. In this approach the database sites are organized into a hierarchy (or tree) and there is a deadlock detector for each node of the hierarchy. For example, one might group sites by

---

*Actually, the DMs need only send changes in their graphs -- i.e., newly created or erased edges -- to the deadlock detector.

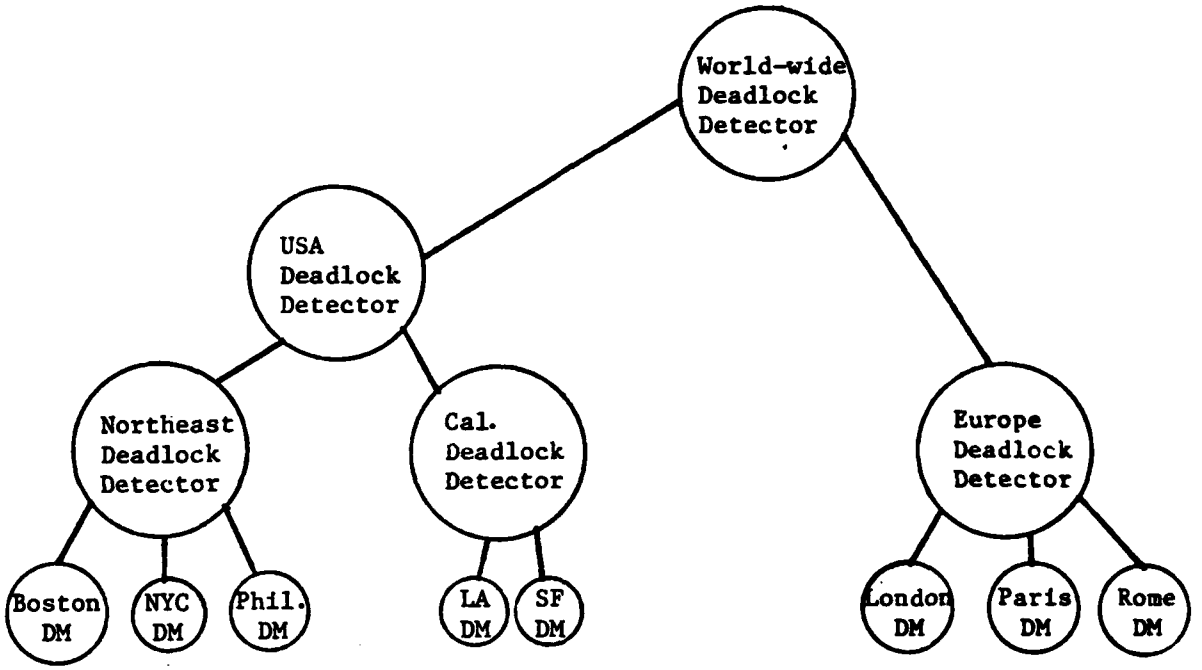
region, then by country, then by continent, etc. Deadlocks that are local to a single site would be detected at that site; deadlocks involving two or more sites of the same region would be detected by the regional deadlock detector, etc.

The hierarchical deadlock detection technique of [MM] is described using a transaction processing model that differs from ours. The technique can be translated into our model in two ways.

In one case the hierarchy is scheduler-based. The leaves of the hierarchy are 2PL schedulers, while interior nodes represent deadlock detectors; see figure 4.4. Every transaction T in the system is identified with a node N of the hierarchy such that all locks owned (or requested) by T are managed by schedulers that are descendants of N; see figure 4.4. Thus, if all locks requested by T are local to one site, T would be identified with that site; if T only requires locks at two sites of the same region, then T would be identified with the region, etc.

Each 2PL scheduler constructs its local waits-for graph as for the centralized technique. Local deadlocks are detected and resolved immediately by each scheduler. To detect non-local deadlocks each scheduler executes the following algorithm periodically. Let G_i be the waits-for graph of scheduler S_i.

DM-based Deadlock Detection Hierarchy Figure 4.4



- If T accesses data at only one DM -- e.g. Boston -- T's "deadlock detector" is the Boston DM.
- If T accesses data at Boston and Phil., its deadlock detector is the Northeast D.D.
- If T accesses data at Boston and SF, its deadlock detector is the USA D.D.
- If T accesses data at Boston and Rome, its deadlock detector is the World-wide D.D.

1. Construct G_i^+, the transitive closure of G_i. G_i^+ contains the same nodes (i.e. transactions) as G_i but more edges. In particular G_i^+ contains an edge <T, T'> iff G_i contains a path from T to T'.
2. Eliminate from G_i^+ all nodes (i.e., transactions) that are identified with scheduler S_i. Let G_i' denote the resulting graph.
3. Construct $(G_i')^r$, the transitive reduction of G_i'. $(G_i')^r$ is a minimal representation of G_i'.
4. Transmit $(G_i')^r$ to the parent of scheduler S_i in the hierarchy.

Each interior node of the hierarchy (i.e., each deadlock detector) receives the reduced waits-for graphs from its children. The deadlock detector constructs the union of these graphs and checks for cycles. If a cycle is found, a deadlock has been detected and is resolved in the standard manner. Otherwise the deadlock detector executes the preceding algorithm and transmits the result to its parent. This procedure continues until the top of the hierarchy is reached.

A similar technique can be devised for a TM-based hierarchy.

Problems with Deadlock Detection

Although centralized and hierarchical deadlock detection differ in detail, both involve the periodic transmission of local

waits-for information to one or more deadlock detector sites. The periodic nature of the deadlock detection process introduces two problems:

1. Because deadlock detection is executed periodically, a deadlock may exist for several minutes without being detected. This can cause substantial response time degradation. The solution is to execute the deadlock detector more frequently. However, this increases the cost of deadlock detection.
2. Suppose a transaction T which owns some locks and is waiting for others is restarted for reasons other than concurrency control (e.g., its site crashed). Until T's restart is propogated to the deadlock detector, the deadlock detector can find a cycle in the waits-for graph that includes T. Such a cycle is called a phantom deadlock. When the deadlock detector discovers a phantom deadlock, it may unnecessarily restart a transaction other than T. The effect of phantom deadlocks is to cause transactions to be restarted unnecessarily. (As in the case of preemptive deadlock prevention, care must be taken not to erroneously restart any transaction in the second phase of two-phase commit).

4.1.9 Deadlock Resolution for Voting 2PL

The voting technique of Section 4.1.4 introduces certain deadlock resolution subtleties. The difficulty lies in defining the waits-for relationship properly in the context of voting.

Suppose logical data item X has three copies -- x_1, x_2, and x_3 -- and suppose transaction T_i owns write-locks on x_1 and x_2 but T_i's lock request for x_3 is blocked by T_j. Insofar as the scheduler for x_3 is concerned, T_i is waiting for T_j. However, since T_i has a majority of the copies locked, the voting protocol permits T_i to proceed without waiting for T_j. This fact should be incorporated into the deadlock resolution scheme to avoid unnecessary restarts.

4.1.10 Heuristics for Reducing Deadlock

This section describes three techniques which attempt to reduce the cost and/or likelihood of deadlock.

Predeclaration of Locks

In the 2PL implementation described so far, locks are requested at the last possible moment. For example, read-locks are requested by dm-read operations; if these locks were requested any later, then the dm-read would violate the specification of 2PL (see Section 4.1.1). Similarly, write-locks are requested by pre-commit operations; if these locks were requested any later, then two-phase commit would not work correctly. (For example, if write-locks were requested by dm-writes, a transaction could deadlock during the second phase of two-phase commit.)

However, if a system is deadlock prone, it may be preferable to request locks earlier. The objective is to force deadlocks to occur earlier in the execution of transactions so that the cost of restarting deadlocked transactions is reduced.

The extreme version of this heuristic calls for transactions to predeclare their locks, meaning that all locks are obtained before the transaction starts its main execution. Predeclaration has two main disadvantages:

1. Before a transaction executes it may be difficult to predict the data it will access. To pre-declare in these cases it is necessary to lock all data the transaction might access.
2. Predeclaration causes locks to be held for a longer period of time than is necessary.

Both of these disadvantages tend to increase the probability that other transactions will be delayed or restarted by the predeclared locks.

Pre-ordering of Resources

A common deadlock avoidance technique in operating systems is to assign numbers to all resources and to require that processes lock resources in numeric order. This technique is not generally applicable to database systems since transactions may request locks at various times during their execution. However, it can be applied if locks are pre-declared. The technique can also be used if write-locks are requested by pre-commits and all pre-commits are issued at the same time (In the latter case, the technique only prevents deadlocks involving ww conflicts; deadlocks involving rw conflicts are still possible).

The principal disadvantage of this technique is that it forces locks to be obtained sequentially. This tends to increase transaction response time.

Re-ordering of Waiting Queues

In the basic 2PL implementation lock requests for a given data item are processed in FIFO order (see Section 4.1.2). In some cases, though, it may be preferable to process requests in other orders. For example, if deadlock prevention is in effect, one

should clearly process requests in priority order. It may also be desirable to process requests for read-locks before write-locks (or vice versa), etc.

These issues are often important in operating systems. However, in centralized database systems the waiting queues for individual data items are usually short, so these issues are normally unimportant. The importance of these factors in distributed database systems is unknown.

4.2 Timestamp Ordering (T/O)

4.2.1 Specification

Timestamp ordering (T/O) is a technique whereby the serialization order is selected a priori and transaction execution is forced to obey this order. This is in sharp contrast to 2PL, where the serialization order is induced during execution by the order in which locks are obtained. In timestamp ordering, each transaction is assigned a unique timestamp by its TM. (The timestamp generation technique of Section 4.1.5 can be used here). The TM attaches the timestamp to all dm-read and dm-write operations issued on behalf of the transaction; DMs are required to process conflicting operations in timestamp order.

The definition of conflicting operations depends on the type of synchronization being performed and is analogous to conflicting locks. For rw synchronization, two operations conflict iff (a) both operate on the same data item x, and (b) one is a dm-read and the other is a dm-write. For ww synchronization, two operations conflict iff (a) both operate on the same data item, and (b) both are dm-writes.

It is easy to prove that T/O is a correct synchronization technique, meaning that T/O attains an acyclic ->rwr (resp. ->ww) relation when used for rw (resp. ww) synchronization. Since each DM processes conflicting operations in timestamp order, each edge of the ->rwr (resp. ->ww) relation is in timestamp order. Consequently, all paths in the relation are in timestamp order and, since all transactions have unique timestamps, it follows that no cycles are possible. In addition, the timestamp order is a valid serialization order.

4.2.2 Basic Implementation

An implementation of T/O amounts to building a T/O scheduler, a software module that receives dm-read and dm-write operations and outputs these operations in accordance with the T/O specification [SM 1,2]. In practice, pre-commits must also be processed through the T/O scheduler for two-phase commit to operate properly. In Sections 4.2.2-4.2.8 we describe T/O implementations without considering the impact of two-phase commit. In Section 4.2.9 we factor two-phase commit into these implementations.

As for 2PL, the basic T/O implementation distributes the schedulers along with the database. Consider the T/O scheduler at some particular DM. For each date item x stored at the DM,

the scheduler keeps track of the largest timestamp of any dm-read that has operated on x; this is called the R-timestamp of x. The W-timestamp of x is defined similarly.

The basic T/O scheduler operates as follows. Assume the scheduler is performing rw synchronization. To process a dm-read(x), the scheduler compares the timestamp of the dm-read to the W-timestamp of x. If the timestamp of the dm-read is larger, the dm-read is output by the scheduler and the R-timestamp of x is updated; the new R-timestamp of x equals the maximum of (a) the old R-timestamp of x, or (b) the timestamp of the dm-read. If the timestamp of the dm-read is smaller than the W-timestamp of x, then the dm-read is rejected and the issuing transaction is aborted.

Similarly, to process a dm-write(x), the scheduler compares the timestamp of the operation to the R-timestamp of x. If the former timestamp is larger, the dm-write is output and the W-timestamp of x is updated to the maximum of (a) the old W-timestamp of x, or (b) the timestamp of the dm-write. Otherwise, the dm-write is rejected and the transaction is aborted.

For ww synchronization, the T/O scheduler operates as follows. To process a dm-write (x), the scheduler compares the timestamp of the dm-write to the W-timestamp of x. If the dm-write has a larger timestamp, the dm-write is output and the W-timestamp of

x is set equal to the timestamp of the dm-write. Otherwise, the dm-write is rejected and the transaction is aborted.

When a transaction is aborted, it is assigned a larger timestamp by its TM and is restarted. This restart policy can lead to a cyclic restart situation (see Section 4.1.5). Cyclic restart can be avoided by assigning an especially large timestamp to the transaction, thereby reducing the probability of a subsequent restart. Other restart policies are discussed in the following sections.

This implementation of T/O requires slightly less communication than 2PL, because locks need not be released. (This remains true even when two-phase commit is considered; see Section 4.2.9). However, this implementation requires a substantial amount of storage for maintaining timestamps. Techniques for reducing this storage requirement are discussed in Section 4.2.8.

4.2.3 The Thomas Write Rule

For ww synchronization the basic T/O scheduler can be optimized using an observation of [Thomas 1,2]. Suppose the timestamp of a dm-write(x) is smaller than the W-timestamp of x. Instead of rejecting the dm-write (and restarting the issuing transaction)

we can simply ignore the dm-write. We call this the Thomas Write Rule (TWR).

Intuitively, TWR only applies to a dm-write that is seeking to place obsolete information into the database. The rule guarantees that the effect of applying a set of dm-writes to x is identical to what would have happened had the dm-writes been applied in timestamp order. Thus, the effect is independent of the order in which the dm-writes are applied. Note that TWR has exactly the same effect as synchronizing dm-writes using the basic T/O scheduler.

4.2.4 Multi-Version T/O

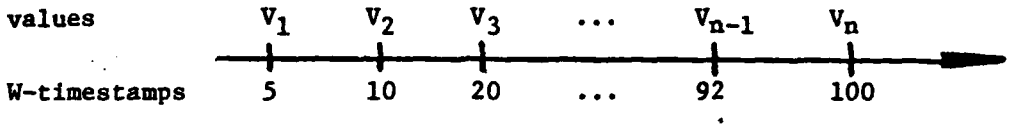
For rw synchronization the basic T/O scheduler can be improved by using the multi-version data item concept of [Reed]. For each data item x we maintain a set of R-timestamps, and a set of <W-timestamp, value> pairs. Intuitively, the R-timestamps of x record the timestamps of all dm-read operations that have ever read x; the <W-timestamp, value> pairs record the timestamps of all dm-writes that have ever written into x, along with the values written. The <W-timestamp, value> pairs are called the versions of x.

Using multi-versions, one can achieve rw synchronization without ever rejecting dm-reads. Consider a dm-read on x with timestamp TS. To process this operation, we simply read the version of x with largest timestamp less than TS; see figure 4.5a. However, dm-writes can still be rejected. Consider a dm-write on x with timestamp TS_1, and let TS_2* be the smallest W-timestamp of x greater than TS_1; see figure 4.5b. If any R-timestamp of x lies between TS_1 and TS_2 then the dm-write is rejected. If no R-timestamp lies in that range, then the scheduler outputs the dm-write; this causes a new version of x to be created with timestamp TS_1.

The correctness of this technique can be proved as follows. Consider a dm-read(x) that is processed "out of order". I.e., suppose the dm-read(x) has timestamp TS_1 yet it is processed after some dm-write(x) with a larger timestamp TS_2. The dm-read ignores all versions of x whose timestamps are larger than TS_1; thus, the value read by the dm-read is identical to the value it would have read had it been processed "in order". Now consider a dm-write(x) that is processed "out of order". I.e., suppose the dm-write has timestamp TS_1, yet it is processed after some dm-read with a larger timestamp TS_2. Since the dm-write was not rejected, there must exist a version of x with timestamp TS_1' such that $TS_1 < TS_1' < TS_2$. Again the effect is identical to the effect of a timestamp ordered execution. Q.E.D.

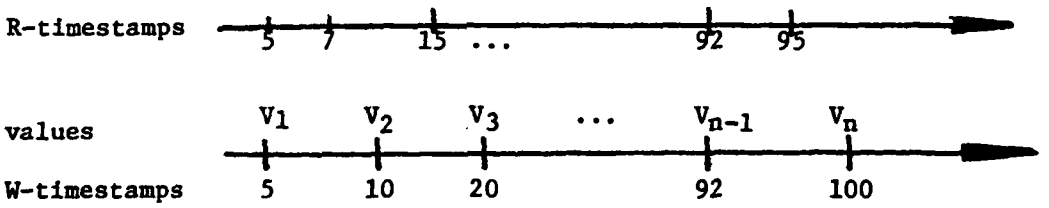
Multi-version Reading and Writing Figure 4.5

a) Let us represent the versions of a data item x on a "time line"

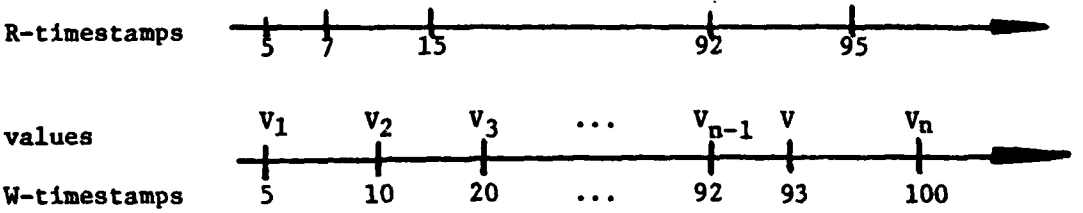


To process a dm-read(x) with timestamp 95, find the biggest W-timestamp less than 95; in this case 92. That is the version you read. So in this case, the value read by the dm-read is V_{n-1}.

b) Let us represent the R-timestamps of x similarly



To process a dm-write(x) with timestamp 93, we create a new version of x with that timestamp.



However, this new version "invalidates" the dm-read of part (a), because if the dm-read had arrived after the dm-write, it would have read value V instead of V_{n-1}. Therefore, we must reject the dm-write.

Notice that the multi-version concept achieves ww synchronization "automatically"; the effect of any sequence of dm-writes is identical to the same dm-writes in timestamp order. Insofar as ww synchronization is concerned, the multi-version approach is therefore an embellished version of the Thomas Write Rule.

It is usually not possible to keep all versions forever, so a technique for forgetting (i.e., deleting) versions is needed. This issue is addressed in Section 4.2.8.

4.2.5 Conservative T/O

Conservative timestamp ordering is a technique for eliminating the possibility of restarts during T/O scheduling. When a scheduler receives an operation O that might cause a future restart, the scheduler delays the processing of O until it is certain that no future restarts are possible.

Imagine that each T/O scheduler has a collection of input queues, one R-queue and one W-queue per TM. Each R-queue represents a FIFO (i.e., pipelined) channel for the transmission of dm-reads from one TM to one scheduler, and each W-queue

---

*TS_2 equals infinity if TS_1 is the largest W-timestamp of x.

represents a pipelined channel for the transmission of dm-writes from one TM to one scheduler. Each TM is required to place operations into any given queue in timestamp order. For example, if TM_i sends dm-read(x) to some scheduler S_j followed by dm-read(y), the timestamp of dm-read(x) must be less than or equal to the timestamp of dm-read(y). Since the queues are pipelined, each scheduler receives dm-read operations from each individual TM in timestamp order. Similarly, each scheduler receives dm-write operations from each individual TM in timestamp order.

This structure can be used to eliminate restarts during rw synchronization as follows. Suppose scheduler S_j wants to output a dm-read(x) with timestamp TS. If S_j outputs this operation too early, it may cause subsequent dm-writes to be rejected. S_j can avoid the rejection of dm-writes by delaying the dm-read until it is certain that it has processed all dm-writes with smaller timestamps. An algorithm that accomplishes this goal for rw synchronization is sketched below.

1. Scan each W-queue. If the first dm-write on any W-queue has timestamp less than TS, output the dm-write.
2. Repeat step 1 until every W-queue is nonempty and the first operation in each W-queue has timestamp greater than TS.

3. Output the dm-read.

This algorithm ensures that S_j will not output the dm-read until it has processed all dm-writes with smaller timestamp. Consequently, S_j is never forced to reject a dm-write.

However, this algorithm is not quite correct as stated. Let T_i be transaction issuing the dm-read with timestamp TS and let TM_i be the TM executing T_i. Step 2 of the algorithm requires waiting until each W-queue (including TM_i's) has a dm-write with timestamp greater than TS. However, if T_i intends to send a dm-write to S_j (with timestamp TS), TM_i cannot allow any dm-writes with timestamps greater than TS to be sent to S_j. And, even if T_i does not intend to send such a dm-write, TM_i may not know this fact at the time it sends T_i's dm-read to S_j. Thus, S_j may be deadlocked waiting for a dm-write from TM_i's W-queue. One solution to this problem is for TM_i to send a message through its W-queue to S_j indicating it has no more dm-writes with timestamps smaller than TS to send to S_j, thereby forcing S_j to output T_i's dm-read. Such messages, called null operations, will be described momentarily.

The algorithm does not, however, prevent the rejection of dm-reads. One way to solve this problem is to use multi-version T/O (see Section 4.2.4). Alternatively, the scheduler can delay the processing of dm-writes until it is certain that it has processed all dm-reads with smaller timestamps using an

algorithm similar to the above. If the latter alternative is adopted, the scheduling algorithm for rw synchronization may be sketched as follows.

1. Initialize R-TS and W-TS to 0. ntuitively R-TS is the smallest timestamp of any pending dm-read, provided there is at least one dm-read on every R-queue. W-TS has a similar meaning.
2. Output all possible dm-writes, as follows.

2.1 Scan each W-queue. If the first dm-write on any queue has timestamp less than R-TS, then output that dm-write.

2.2 Repeat 2.1 until no further dm-writes can be output.

2.3 If any W-queue is empty, then let W-TS := 0. Otherwise let W-TS := the minimum timestamp of any queued dm-write.

3. Output all possible dm-reads, as follows.

3.1 Scan each R-queue. If the first dm-read on ay queue has timestamp less than W-TS, then output that dm-read.

3.2 Repeat 3.1 until no further dm-reads can be output.

3.3 If any R-queue is empty, then let R-TS := 0. Otherwise let R-TS := the minimum timestamp of any queued dm-read.

4. Go to step 2. When a dm-read (resp. dm-write) arrives at the only empty R-queue (resp. W-queue), then change R-TS (resp. W-TS) (which is currently 0) to be the minimum timestamp of ay queued dm-read (resp. dm-write).

The algorithm for ww synchronization is simpler. In this case, the scheduler need only wait until every W-queue is nonempty.

The scheduler then outputs the dm-write with smallest timestamp. If conservative T/O is used for both rw and ww synchronization, the algorithm is equaliy simple. In this case, the scheduler waits until every queue is nonempty and then outputs the operation with smallest timestamp.

The above implementation of conservative T/O suffers three major problems.

1. The implementation does not guarantee termination -- if some TM never sends an operation to some scheduler, the scheduler will "get stuck" due to the empty queue and will never output any operations.
2. The implementation requires that all TMs communicate regularly with all schedulers -- this is infeasible in large networks.
3. The implementation is overly conservative -- e.g., the combined rw and ww algorithm processes all operations in timestamp order, not merely conflicting operation.

Problems 1 and 2 are addressed below. Problem 3 is considered in Section 4.2.6.

Guaranteeing Termination -- Null operations

To guarantee termination, we require that TMs periodically send timestamped null-operations to each scheduler, in the absence of any "real" traffic. A null-operation is a dm-read or dm-write

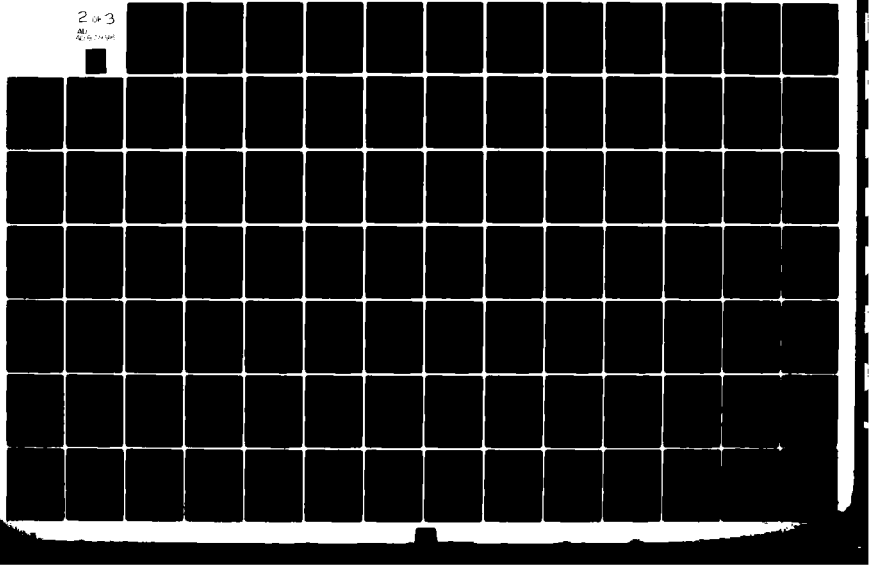
that does not reference a data item. Its purpose is to convey timestamp information to the scheduler. A null-dm-read (resp. null-dm-write) operation with timestamp TS sent from TM_i to scheduler S_j tells S_j that TM_i will not send it any more dm-read 8resp.dm-write) operations with timestamps smaller than TS. Thus, any scheduling decision that required S_j's receiving all dm-reads (resp. dm-writes) from TM_i timestamped less than TS can be made after that null-dm-read (resp. null-dm-write) is received.

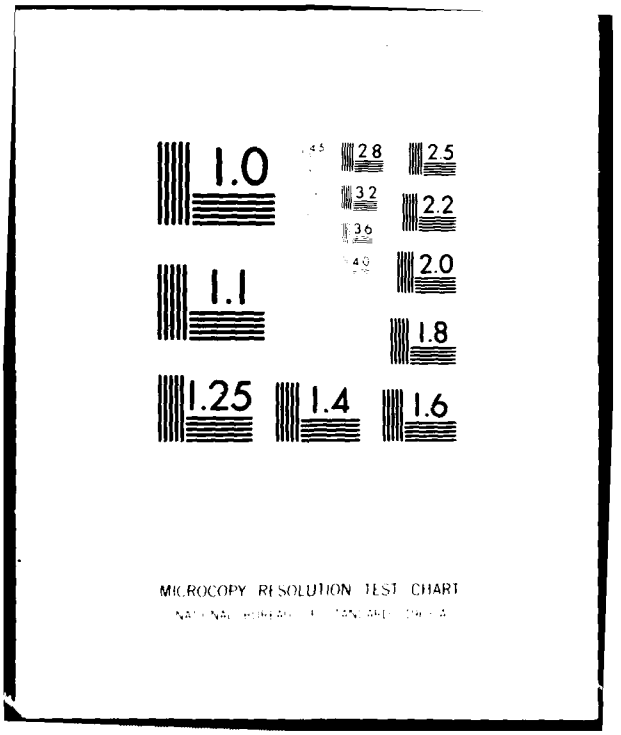
An impatient scheduler can prompt a TM for a null-operation by sending a request-null operation to it. The request-null should specify the type of null-operation desired (read or write) and the timestamp the scheduler is waiting for. For example, suppose conservative T/O is being used for rw synchronization, and suppose S_j wants to process a dm-read with timestamp TS. To process this dm-read, S_j must wait until each W-queue contains a dm-write with timestamp greater than TS. If the W-queue from TM_i does not satisfy this property, S_j may send TM_i a request-null requesting a null-dm-write with timestamp greater than TS.

Avoiding Unnecessary Communication

To avoid unnecessary communication between TMs and schedulers, null-operations with very large timestamps can be used. For example, if TM_i rarely needs to access the database protected by

AD-A087 996 COMPUTER CORP OF AMERICA CAMBRIDGE MA F/G 9/2
FUNDAMENTAL ALGORITHMS FOR CONCURRENCY CONTROL IN DISTRIBUTED D--ETC(U)
MAY 80 P A BERNSTEIN, N GOODMAN F30602-79-C-0191
UNCLASSIFIED RADC-TR-80-158 NL




1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6
MICROCOPY RESOLUTION TEST CHART

S_i, TM_i should send null-operations to S_j with timestamps far in the future, e.g., every hour TM_i could send S_j a null-operation whose timestamp equals the current time plus one hour. Of course, if TM_i needs to send a "real" dm-read to S_j at some time during the hour, a mechanism is required to retract the large timestamp and replace it by a more reasonable one.

In extreme cases, TM_i can send S_j a null-operation with infinite timestamp. This signifies that TM_i does not intend to communicate with S_j until further notice.

4.2.6 Conservative T/O with Transaction Classes

Another technique for reducing communication is transaction classes [BRGP]. As in lock predeclaration (Section 4.1.9), we assume that the readset and writeset of every transaction is known in advance. This information is used to group transactions into predefined classes. Class definitions are used to support a less conservative scheduling policy.

A transaction class is defined by a readset and a writeset. A transaction T is a member of class C iff T's readset is a subset of C's readset, and T's writeset is a subset of C's writeset. Classes need not be disjoint; i.e., T may be a member of several classes. Figure 4.6 illustrates these definitions.

that does not reference a data item. Its purpose is to convey timestamp information to the scheduler. A null-dm-read (resp. null-dm-write) operation with timestamp TS sent from TM_i to scheduler S_j tells S_j that TM_i will not send it any more dm-read 8resp.dm-write) operations with timestamps smaller than TS. Thus, any scheduling decision that required S_j's receiving all dm-reads (resp. dm-writes) from TM_i timestamped less than TS can be made after that null-dm-read (resp. null-dm-write) is received.

An impatient scheduler can prompt a TM for a null-operation by sending a request-null operation to it. The request-null should specify the type of null-operation desired (read or write) and the timestamp the scheduler is waiting for. For example, suppose conservative T/O is being used for rw synchronization, and suppose S_j wants to process a dm-read with timestamp TS. To process this dm-read, S_j must wait until each W-queue contains a dm-write with timestamp greater than TS. If the W-queue from TM_i does not satisfy this property, S_j may send TM_i a request-null requesting a null-dm-write with timestamp greater than TS.

Avoiding Unnecessary Communication

To avoid unnecessary communication between TMs and schedulers, null-operations with very large timestamps can be used. For example, if TM_i rarely needs to access the database protected by

Transaction Classes
Figure 4.6

- A class is defined by a readset and a writeset. E.g.

C_1: readset = $\{x_1\}$, writeset = $\{y_1, y_2\}$

C_2: readset = $\{x_1, y_2\}$, writeset = $\{y_1, y_2, z_2, z_3\}$

C_3: readset = $\{y_2, z_3\}$, writeset = $\{x_1, z_2, z_3\}$

- A transaction is a member of a class if its readset is a subset of the class readset and its writeset is a subset of the class writeset. E.g.

T_1: readset = $\{x_1\}$, writeset = $\{y_1, y_2\}$

T_2: readset = $\{y_2\}$, writeset = $\{z_2, z_3\}$

T_3: readset = $\{z_3\}$ writeset = $\{x_1\}$

- T_1 is a member of C_1 and C_2

- T_2 is a member of C_2 and C_3

- T_3 is a member of C_3

Transaction classes are defined statically meaning that class definitions are not expected to change frequently during normal operation of the system. Changing a class definition is akin to changing the database schema and requires mechanisms that are beyond the scope of this report. We assume that class definitions are stored in static tables which are readily available at any site that requires this information.

Classes are associated with TMs. Every transaction that executes at a TM must be a member of a class associated with the TM. If a transaction is submitted to a TM at which this property does not hold, the transaction is forwarded to another TM at which an appropriate class exists.

For notational convenience, we assume that every class is associated with exactly one TM, and conversely, every TM is associated with exactly one class. We use C_i to denote the class associated with TM_i. This notation simplifies our discussion, but does not constrain system operation in any way. For example, suppose we wish to execute transactions that are members of class C_1 at two TMs, say TM_1 and TM_2. To do so, we merely define another class C_2 with the same readset and writeset as C_1 and associate C_1 with TM_1 and C_2 with TM_2. On the other hand, suppose we wish to execute transactions that are members of two classes, say C_1 and C_2, at one TM. To do so, we merely multi-program another TM at the same site as the first.

Transaction classes are exploited by conservative T/O schedulers as follows. Consider rw synchronization and suppose that scheduler S_j wants to output a dm-read(x) with timestamp TS. Instead of waiting for dm-writes with smaller timestamp from all TMs, S_j need only wait for dm-writes from those TMs whose class writeset contains x. Similarly, to process a dm-write(x) with timestamp TS, S_j need only wait for dm-reads with smaller timestamp from those TMs whose class readset contains x. Thus, the level of concurrency in the system is increased. ww synchronization proceeds in a similar fashion.

This technique also reduces communication requirements, since a TM need only communicate with a scheduler if its class readset or writeset contains data items protected by the scheduler.

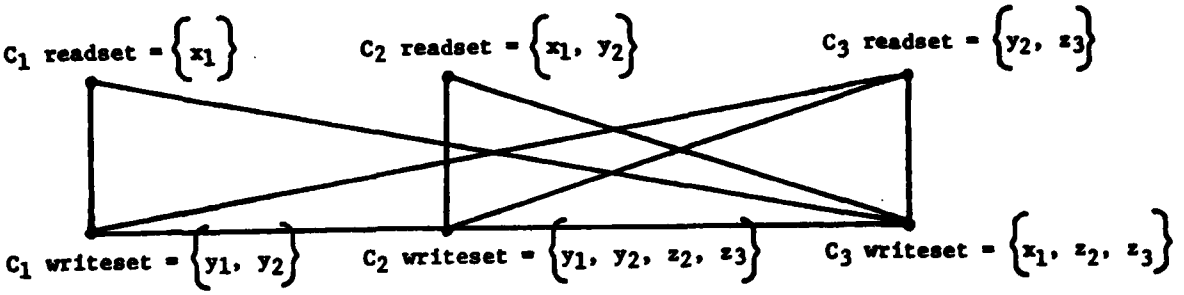
4.2.7 Conservative T/O with Conflict Graph Analysis

Conflict graph analysis is a technique for further improving the performance of conservative T/O with classes. A conflict graph is an undirected graph that summarizes potential conflicts between transactions in different classes. For each class C_i the graph contains two nodes, denoted r_i and w_i. Intuitively, these nodes represent the readset and writeset of C_i. The edges of the graph are defined as follows (refer to figure 4.7).

Conflict Graph

Figure 4.7

Define C_1, C_2, C_3 as in figure 4.6



(i) For each class C_i, there is an edge between r_i and w_i; this edge is called a vertical edge.

(ii) For each pair of classes C_i and C_j (with $i \neq j$) there is a horizontal edge between w_i and w_j iff the writeset of C_i intersects the writeset of C_j.

(iii) For each pair of classes C_i and C_j (with $i \neq j$) there is a diagonal edge between r_i and w_j iff the readset of C_i intersects the writeset of C_j.

Intuitively, a horizontal edge indicates a condition in which a scheduler S_k may be forced to delay dm-write operations for purposes of ww synchronization. Suppose classes C_i and C_j are connected by a horizontal edge (i.e., there is an edge between w_i and w_j). This means that the class writesets intersect. So, if S_k receives a dm-write from C_i, it may be necessary for S_k to delay the dm-write until S_k receives all dm-writes with smaller timestamps from C_j.

Similarly, a diagonal edge indicates that S_k may be forced to delay operations for rw synchronization. Suppose C_i and C_j are connected by a diagonal edge -- in particular, assume that r_i is connected to w_j. This means that the readset of C_i intersects the writeset of C_j. So, if S_k receives a dm-read from C_i, it may be necessary to delay the dm-read until all dm-writes from C_j with smaller timestamp are received. Symmetrically, if S_k

receives a dm-write from C_j, this operation will be delayed until sufficient dm-reads are received from C_i.

Conflict graph analysis improves the situation by identifying inter-class conflicts that cannot cause non-serializable behavior. This corresponds to identifying horizontal and diagonal edges that do not require synchronization. In particular, it is proved in [BS 2] that schedulers need only synchronize dm-writes from C_i and C_j if either

1. the horizontal edge between w_i and w_j is embedded in a cycle of the conflict graph; or
2. portions of the intersection of C_i's writeset and C_j's writeset are stored at two or more DMs.

In other words, if conditions (1) and (2) do not hold, a scheduler S_k need not process dm-writes from C_i and C_j in timestamp order.

Similarly, it is proved in [BS 2] that dm-reads from C_i and dm-writes from C_j need only be processed in timestamp order if either

1. the diagonal edge between r_i and w_j is embedded in a cycle of the conflict graph; or
2. portions of the intersection of C_i's readset and C_j's writeset are stored at two or more DMs.

Since classes are defined statically, conflict graph analysis is also performed statically. The output of this analysis is a table indicating which horizontal and vertical edges require synchronization and which do not. This information, like class definitions, is distributed in advance to all schedulers that require it.

Conservative T/O with conflict graph analysis has been implemented in the SDD-1 distributed database system [BSR].

In principle, conflict graph analysis can be applied to other synchronization techniques to improve their performance as well. Theoretical aspects of this integration are examined in [BSW], but many details remain to be worked out.

4.2.8 Timestamp Management

A common criticism of T/O schedulers is that too much memory is needed to store timestamps. This problem can be overcome by "forgetting" old timestamps.

Timestamps are used in basic T/O to reject operations that "arrive late", e.g., to reject a dm-read(x) with timestamp TS_1 tnat arrives after a dm-write (x) with timestamp TS_2, where $TS_1 < TS_2$. In principle, TS_1 and TS_2 can differ by an arbitrary amount. However, in practice it is unlikely that these timestamps will differ by more than a few minutes.

For example, suppose timestamps are generated using local real-time clocks as described in Section 4.1.6. It is reasonable to assume that these clocks can be kept approximately synchronized, say to within 5 minutes of each other, using, say, the technique of [Lamport 1]. Assume further that once a transaction is assigned a timestamp, it completes execution within 5 minutes and that the network delivers all operations within 5 minutes. Given these assumptions, it is unlikely that a scheduler will receive an operation whose timestamp is more than 15 minutes behind real-time. Consequently there is little point in storing older timestamps.

Because of this observation, timestamps can be stored in relatively small tables which are periodically purged. R-timestamps are stored in a table called the R-table. Entries in the table are ordered pairs of the form <x, R-timestamp>; for any data item x, there is at most one entry. In addition, there is a variable, R-min, which tells the maximum value of any timestamp that has been purged from the table.

When the scheduler needs to know the R-timestamp of x, it searches the R-table for an <x, TS> entry. If such an entry is found, then TS is the correct R-timestamp of x. Otherwise, the R-timestamp of x is less-than-or-equal-to R-min. To err on the side of safety, the scheduler assumes that the R-timestamp of x equals R-min. To update the R-timestamp of x, the scheduler

modifies the <x, TS> entry, if one exists. Otherwise, a new entry is created and added to the table.

When the R-table is full, the scheduler selects an appropriate value for R-min and deletes all entries from the table with smaller timestamp.

W-timestamps are managed by a similar discipline, and analogous techniques can be devised for multi-version databases.

The timestamp situation for conservative T/O is even better, because conservative T/O only requires timestamped operations, not timestamped data. If conservative T/O is used for rw synchronization, the R-timestamps of data items are rendered useless and may be discarded. If conservative T/O is used for both rw and ww synchronization, then W-timestamps can be eliminated as well.

4.2.9 Integrating Two-Phase Commit into T/O

It is necessary to integrate two-phase commit into the T/O implementations described above in order to ensure atomic commitment of updates (see Section 2). This is done by timestamping pre-commit operations and modifying the T/O implementations to accept or reject pre-commits instead of dm-writes. If a scheduler rejects a pre-commit, the issuing

transaction is aborted. However, if a scheduler accepts a pre-commit, it must guarantee to accept the corresponding dm-write no matter when that operation arrives. To make this guarantee, the scheduler may be forced to delay conflicting operations that arrive before the dm-write.

In the rest of this section we explore the impact of two-phase commit on each of the T/O implementations described previously.

Integrating Two-Phase Commit Into Basic T/O

Consider a pre-commit(x) with timestamp TS. Let P denote this operation and let W denote the corresponding dm-write.

Assume that basic T/O is being used for rw synchronization. P can be accepted by a scheduler iff TS is greater than the R-timestamp of x; i.e., P is accepted iff the scheduler can still output W. Once the scheduler accepts P, it must guarantee that TS will remain greater than the R-timestamp of x until W is received. To make this guarantee, the scheduler simply refuses to output any dm-reads on x with timestamps greater than TS, until W is received. If any such dm-reads arrive before W, the scheduler places them on a waiting queue. The effect is similar to setting a write-lock on x for the duration of two-phase commit.

The algorithm for ww synchronization is similar. In this case, P is accepted by the scheduler iff TS is greater than the

W-timestamp of x. Once the scheduler accepts P, it agrees not to output any dm-writes on x with timestamps greater than TS, until it receives W. If any such dm-writes arrive before W, the scheduler places them on a waiting queue as above. The effect differs from setting a write-lock on x in that pre-commits on x with timestamps greater than TS can be accepted before W arrives.

Integrating Two-Phase Commit Into Thomas Write Rule

The Thomas Write Rule applies only to ww synchronization and eliminates the possibility of rejecting a dm-write for purposes of ww synchronization. If the Thomas Write Rule is in effect, there is no need to incorporate two-phase commit into the ww synchronization algorithm. Pre-commits must still be sent to all sites that are being updated, but the pre-commits need not be processed by the ww scheduler.

Integrating Two-Phase Commit Into Multi-Version T/O

Like the Thomas Write Rule, the multi-version concept eliminates the need to consider two-phase commit insofar as ww synchronization is concerned. However, two-phase commit remains as issue for rw synchronization.

Let P be a pre-commit(x) with timestamp TS_1 and let W be the corresponding dm-write. When P arrives at a scheduler, the

scheduling rule of Section 4.2.4 is applied: let TS_2 be the smallest W-timestamp of x greater than TS_1; if any R-timestamp of x lies between TS_1 and TS_2, then P is rejected, otherwise P is accepted. If the scheduler accepts P, it agrees not to output any dm-reads on x with timestamps between TS_1 and TS_2 until W is received. As in the preceding algorithms, all such dm-reads that arrive before W are placed on a waiting queue.

Integrating Two-Phase Commit Into Conservative T/O

It is not essential that two-phase commit be tightly integrated into conservative T/O, because conservative T/O never rejects dm-writes. However, scheduling delay can be reduced by transmitting pre-commits via W-queues instead of using these queues for dm-writes.

For example, suppose conservative T/O is being used for rw synchronization, and suppose scheduler S_j wants to output a dm-read(x) with timestamp TS. S_j need only delay this dm-read until each W-queue contains a pre-commit with timestamp greater than TS; there is no need to wait for the corresponding dm-writes. Of course, the dm-read may have to wait for some dm-writes with smaller timestamp; i.e., if S_j has accepted a pre-commit(x) with timestamp TS' < TS, the dm-read cannot be output until the dm-write(x) with timestamp TS' is received. However, this delay is due to two-phase commitment requirements and is independent of the conservative T/O scheduling rule.

If pre-commits are substituted for dm-writes in the W-queues, then null-dm-writes must be replaced by null-pre-commits that serve the same function as null operations carrying a timestamp.

4.2.10 Heuristics for Reducing Restarts

This section describes three heuristics which attempt to reduce the cost or probability of restarts for non-conservative T/O implementations.

Predeclaration of Readsets and Writesets

To reduce the cost of restarts, transactions are advised to issue their dm-reads and pre-commits as early as possible. The extreme version of this heuristic calls for transactions to predeclare their readsets and writesets. This means that dm-reads are issued for the entire readset and pre-commits are issued for the entire writeset before a transaction begins its main execution. If any operation is rejected, the transaction starts over with a larger timestamp. Otherwise, the transaction is guaranteed to execute with no danger of restart.

Delaying of Operations

To reduce the probability of restart, a scheduler can delay the processing of operations to wait for "earlier" operations (i.e.,

ones with smaller timestamps) to arrive. This heuristic is essentially a compromise between conservative and non-conservative T/O, and trades response time for a reduction in restart probability. The amount of delay can be tuned to optimize this trade-off.

Reading Old Versions

The performance of multi-version T/O can be improved by permitting queries (i.e., read-only transactions) to read old versions of data items. Recall that in multi-version T/O, dm-read operations are never rejected, but they may cause subsequent pre-commits to be rejected. For example, once a dm-read(x) with timestamp TS is processed, a subsequent pre-commit(x) with timestamp TS', where TS' < TS, may be rejected.

We can reduce the probability of a pre-commit being rejected by letting queries issue dm-reads with small timestamps. Consider a query Q. Instead of assigning Q a timestamp in the usual way -- i.e., by reading the system clock -- we may assign Q an older timestamp. This reduces the probability that Q will interfere with an active update transaction, but of course, also causes Q to read older data. If Q's timestamp is too old, the data it reads may be obsolete. Thus, this technique entails a compromise between system performance and timeliness of data.

Little is known about this tradeoff in general. However, because the real-world changes so slowly relative to computer speeds, a good compromise should be achievable in many cases. For example, if queries are assigned timestamps that are five minutes old, we would expect few queries to interfere with updates. And in many applications, five minute old data is perfectly acceptable.

As a fringe benefit, this technique also improves the response time for queries by reducing the probability that a query's dm-read operations will be blocked by pre-commits.

5. Integrated Concurrency Control Methods

An integrated concurrency control method consists of two components -- an rw synchronization technique and a ww synchronization technique -- plus an appropriate interface between the components. In this section we list 48 concurrency control methods that can be constructed using the techniques of Section 4 as basic components. Our list of methods is by no means exhaustive. Each method can be further refined by other techniques described in Section 4, bringing the total number of possible concurrency control methods numbers well into the thousands.

Approximately 20 concurrency control methods have been described in the literature. Virtually all of these methods use a single synchronization technique -- either 2PL or T/O -- for both rw and ww synchronization. Indeed, most methods use the same variation of a single technique for both kinds of synchronization. However, such homogeneity is neither necessary nor especially desirable.

For example, several references* propose methods in which basic 2PL is used for both rw and ww synchronization. Other references recommend primary copy 2PL for both tasks. (Later in

this section we shall identify which references recommend each method.) No one has yet proposed a method in which these synchronization techniques are combined. However the analysis of Section 4.1.3 suggests that a method using basic 2PL for rw synchronization and primary copy 2PL for ww synchronization might be superior to either homogeneous techniques.

More outlandish combinations may offer even better performance. For example, it is possible to combine basic 2PL (for rw synchronization) with the Thomas Write Rule (for ww synchronization). This method has the property that ww conflicts never cause a transaction to be delayed or restarted; i.e., this method permits multiple transactions to write into the same data items concurrently. The details of this method are non-trivial and are discussed in Section 5.3.

Combined methods such as these have not appeared in the literature. A major benefit of this study is to bring such methods to attention. In addition, this analysis emphasizes that each method can be fine-tuned by many other options. For example, every method that includes a 2PL component can be tuned by the choice of deadlock resolution technique; T/O methods may choose to use the "delay" heuristic of Section 4.2.10; etc. These refinements may have a substantial impact on the overall performance of the method.

In this section we list the 48 concurrency control methods that we deem to be principal methods. The choice of methods to include here is based on structural and expository considerations, and is arbitrary to some extent. The methods we do not include are deemed to be refinements of the the principal methods.

In Section 5.1 we list and describe 12 methods that use 2PL techniques for both rw and ww synchronization. In Section 5.2 we describe 12 methods that use T/O techniques for both kinds of synchronization. The concurrency control methods in these sections are easy to describe given the material of Section 4: the description of each method is little more than a description of each component technique. Consequently, these sections are repetitive within themselves and relative to Section 4. We include this material to make our analysis of concurrency control more concrete.

In Section 5.3 we list 24 concurrency control methods that combine 2PL and T/O techniques. As we will illustrate in Section 5.3, methods of this type can exhibit useful properties that cannot be attained by pure 2PL or T/O methods.

5.1 Pure 2PL Methods

The 2PL synchronization techniques of Section 4 can be integrated to form twelve principal 2PL methods, listed below.

#	rw technique	ww technique
1	basic 2PL	basic 2PL
2	basic 2PL	primary copy 2PL
3	basic 2PL	voting 2PL
4	basic 2PL	centralized 2PL
5	primary copy 2PL	basic 2PL
6	primary copy 2PL	primary copy 2PL
7	primary copy 2PL	voting 2PL
8	primary copy 2PL	centralized 2PL
9	centralized 2PL	basic 2PL
10	centralized 2PL	primary copy 2PL
11	centralized 2PL	voting 2PL
12	centralized 2PL	centralized 2PL

Each method can be further refined by the choice of deadlock resolution technique (see Section 4.1.6 - 4.1.9), and the deadlock reduction heuristics of Section 4.1.10.

The interface between each 2PL rw technique and each 2PL ww technique is straightforward. The interface need only guarantee

that "two-phased-ness" is preserved. That is, all locks needed for the rw technique and all locks needed for the ww technique must be obtained before any lock is released by either technique.

Sections 5.1.1 - 5.1.3 describe the principal 2PL methods.

5.1.1 Methods Using Basic 2PL for rw Synchronization

Methods 1-4 use basic 2PL for rw synchronization. Consider a logical data item X with copies $x_1,...,x_m$. To read X, a transaction sends a dm-read to any DM that stores a copy of X. This dm-read implicitly requests a read-lock on the copy of X at that DM. To write X, a transaction sends pre-commit operations to every DM that stores a copy of X. These pre-commits implicitly request write-locks on the corresponding copies of X. For all four methods, these write-locks conflict with read-locks on the same copy. These write-locks may also conflict with other write-locks on the same copy, depending on the specific ww synchronization technique used by the method.

Since locking conflict rules for write-locks will vary from copy to copy, we distinguish three types. An rw write-lock only conflicts with read-locks on the same data item. A ww write-lock only conflicts with ww write-locks on the same data

item. And, an rww write-lock conflicts with read-locks, ww write-locks and rww write-locks. Thus, using basic 2PL for rw synchronization, a pre-commit sets rw write-locks.

Method 1 -- Basic 2PL for ww synchronization. In this method, a pre-commit issues rww write-lock (i.e., all write-locks are rww write-locks). Thus, for i=1,...,m a write-lock on x_i conflicts with either a read-lock or a write-lock on x_i.

Method 1 is the "standard" implementation of 2PL in a distributed environment. It has been recommended in one form or another by many authors, including [Gray, RSL, Stonebraker]. Method 1 is also the concurrency control method used by System R* [Selinger].

Method 2 -- Primary copy 2PL for ww synchronization. In this method write-locks only conflict on the primary copy. Let x_1 be the primary copy of X. Then, an rww write-lock is used on x_1, while for i=2,...,m an rw write-lock is used on x_i.

Method 3 -- Voting 2PL for ww synchronization. When voting is used, a DM responds to a pre-commit(x_i) by attempting to set an rww write-lock on x_i. However, if some other transaction already owns an rww write-lock on x_i, then the DM merely sets an rw write-lock. A transaction can write into any copy of X as soon as it obtains rww write-locks on a majority of copies.

Method 4 -- Centralized 2PL for ww synchronization. This method requires that, to write into X, a transaction must first explicitly request a ww write-lock on X from a centralized 2PL scheduler. The rw write-locks set by pre-commit operations never conflict with each other.

In all four methods, read-locks are explicitly released by lock-release operations while write-locks are implicitly released by dm-writes. The lock-release operations may be transmitted in parallel with the dm-writes. In Method 4, additional lock-releases must be sent to the centralized scheduler to release write-locks held there. These operations must be transmitted after all dm-writes have been executed.

5.1.2 Methods Using Primary Copy 2PL for rw Synchronization

Methods 5-8 use primary copy 2PL for rw synchronization. Consider a logical data item X with copies $x_1,...,x_m$, and assume that X_1 is the primary copy. To read X, a transaction must obtain a read-lock on x_1. It may obtain this lock by issuing a dm-read(x_1). Alternatively, the transaction can send an explicit lock-request to x_1's DM; when the lock is granted the transaction can read any copy of X.

To write into X, a transaction must send pre-commits to every DM that stores a copy of X. These pre-commits are processed differently depending on which copy of X is involved. A pre-commit(X_1) implicitly requests an rw write-lock. Pre-commits on other copies of X may also request write-locks depending on the ww technique.

Method 5 -- Basic 2PL for ww synchronization. Every pre-commit requests a ww write-lock. For i=2,. ..,m, the pre-commit(x_i) requests a ww write-lock. Since the write-lock on x_1 must also conflict with read-locks on x_1, pre-commit(x_1) requests an rww write-lock.

Method 6 -- Primary copy 2PL for ww synchronization. The pre-commit(x_1) requests an rww a write-lock on x_1. Pre-commits on other copies do not request any locks.

This method was originally proposed by [Stonebraker] and is the concurrency control method used by Distributed INGRES [SN].

Method 7 -- Voting 2PL for ww synchronization. When a scheduler receives a pre-commit(x_i) for i≠1, it attempts to set a ww write-lock on x_i. When it receives a pre-commit(x_1), it tries to set an rww write-lock on x_1; if it cannot, then the pre-commit should still set an rw write-lock on X_1 (if possible) before waiting for the ww write-lock. A transaction can write into every copy of X as soon as it obtains a ww (or rww) write-lock on a majority of copies of X.

Method 8 -- Centralized 2PL for ww synchronization. This method requires that transactions obtain ww write-locks from a centralized 2PL scheduler. Thus, a pre-commit(x_1) requests an rw write-lock on x_1; for i=2,...,m pre-commit(x_i) does not request any lock at all.

Lock releases for Methods 5-8 are handled as per Section 5.1.1.

5.1.3 Methods Using Centralized 2PL for rw Synchronization

The remaining 2PL methods use centralized 2PL for rw synchronization. Before reading any copy of logical data item X, a transaction must obtain a read-lock on X from a centralized 2PL scheduler, and before writing into X, the transaction must obtain an rw write-lock on X from the centralized scheduler. The transaction must also send pre-commits to every DM that stores a copy of X. Some of these pre-commits implicitly request ww write-locks on copies of X, depending on the specific method.

Method 9 -- Basic 2PL for ww synchronization. Every pre-commit requests a ww write-lock.

Method 10 -- Primary copy 2PL for ww synchronization. Let x_1 be the primary copy of X. A pre-commit(x_1) requests a ww write-lock. Pre-commits on other copies do not request any write-locks.

Method 11 -- Voting 2PL for ww synchronization. Every pre-commit attempts to set a ww write-lock. A transaction can write into every copy of X iff it obtains ww write-locks on a majority of copies of X.

Method 12 -- Centralized 2PL for ww synchronization. In this method, all locks are obtained at the centralized 2PL scheduler. Before writing into a copy of X, an rww write-lock on X is obtained. Pre-commits set no locks at all.

Method 12 is the "standard" implementation of centralized 2PL for a distributed database. It is identical to the primary site method of [AD].

Lock releases for Method 9-12 are handled as per Section 5.1.1.

5.2 Pure T/O Methods

The T/O synchronization techniques of Section 4 can also be integrated to form twelve principal T/O methods. These are

#	rw technique	ww technique
1	basic T/O	basic T/O
2	basic T/O	Thomas Write Rule (TWR)
3	basic T/O	multi-version T/O
4	basic T/O	conservative T/O
5	multi-version T/O	basic T/O

6	multi-version T/O	TWR
7	multi-version T/O	multi-version T/O
8	multi-version T/O	conservative T/O
9	conservative T/O	basic T/O
10	conservative T/O	TWR
11	conservative T/O	multi-version T/O
12	conservative T/O	conservative T/O

(The fact that the number of T/O methods equals the number of 2PL methods is a coincidence).

Each T/O method that incorporates a non-conservative component can be further refined by including (1) techniques for forgetting timestamps (see Section 4.2.8) and (2) heuristics for reducing restarts (see Section 4.2.10). Each method that incorporates a conservative component may also incorporate classes (see Section 4.2.6) and conflict graph analysis (see Section 4.2.7).

The interface between the rw and ww synchronization techniques is even simpler for T/O methods than for 2PL. The only requirement is that both techniques use the same timestamp, i.e., if transaction T has timestamp TS, then TS is used for both rw and ww synchronization.

Sections 5.2.1 - 5.2.3 describe the twelve principal T/O Methods.

5.2.1 Methods Using Basic T/O for rw Synchronization

Methods 1-4 use basic T/O for rw synchronization. Each stored data item in the system, e.g. x_i, has an R-timestamp and a W-timestamp. (Recall from Section 4.2.2 that the R-timestamp (resp. W-timestamp) of x_i is the largest timestamp of dm-read (resp. dm-write) that has read from (resp. written into) x_i). Let T be a transaction with timestamp TS. To read x_i, T issues a dm-read on x_i with timestamp TS; this dm-read is accepted iff TS is greater-than the W-timestamp of x_i. To write into x_i, T issues a pre-commit(x_i) with timestamp TS; this pre-commit is accepted iff (a) TS is greater-than the R-timestamp of x_i, and (b) a condition determined by the ww synchronization technique is also satisfied.

Method 1 -- Basic T/O for ww synchronization. The pre-commit is accepted iff TS is greater-than the R-timestamp and W-timestamp of x_i.

Method 2 -- TWR for ww synchronization. The pre-commit is accepted iff TS is greater-than the largest R-timestamp of x_i. However, if the pre-commit is accepted and TS is less-than the W-timestamp of x_i, then the corresponding dm-write is ignored, i.e., it has no effect on the database.

This method represents an optimization of Method 1 that is apparently preferable in most situations.

Method 3 -- Multi-version T/O for ww synchronization. The pre-commit is accepted iff TS is greater-than the R-timestamp of x_i; the W-timestamp is irrelevant. If the pre-commit is accepted, the corresponding dm-write creates a new version of x_i.

At first glance, this method appears to be a space-inefficient version of Method 2. However this method can be adapted to yield better performance by letting queries read old versions of data items; see Section 4.2.10.

Method 4 -- Conservative T/O for ww synchronization. In this method, pre-commits are processed by each scheduler in timestamp order. I.e., a scheduler S will not process a pre-commit with timestamp TS until it has processed all pre-commits with smaller timestamp, and no pre-commits with larger timestamp. When S processes a pre-commit(x_i) with timestamp TS, it accepts the pre-commit iff TS is greater-than the R-timestamp of x_i.

At first glance this method appears to be a time-inefficient version of Method 2. However, unlike Method 2, this method applies updates to each DM in timestamp order. Consequently, the database at each DM is always consistent in between updates, a property which may be useful for reliability reasons.

5.2.2 Methods Using Multi-version T/O for rw Synchronization

Methods 5-8 use multi-version T/O for rw synchronization. The reader is referred to Section 4.2.4 for a description of this technique. Let T be a transaction with timestamp TS. To read x_i, T issues a dm-read on x_i with timestamp TS; this dm-read is always accepted. To write into x_i, T issues a pre-commit(x_i) with timestamp TS; this pre-commit is accepted iff (a) there is no R-timestamp for x_i that lies between TS and the smallest W-timestamp of x_i larger than TS, and (b) a condition determined by the ww synchronization technique is also satisfied. It is important that once a pre-commit(x) with timestamp TS is accepted, no dm-read(x_i) with larger timestamp be processed until the dm-write(x_i) corresponding to the pre-comit is installed (see Section 4.2.9).

Method 5 -- Basic T/O for ww synchronization. For basic T/O, condition (b) requires that TS be greater-than the largest W-timestamp of x_i. Thus, for Method 5, conditions (a) and (b) may be simplified as follows. The pre-commit is accepted iff TS is greater-than the largest R-timestamp of x_i and the largest W-timestamp of x_i. If the pre-commit is accepted, the corresponding dm-write creates a new version of x_i.

Method 6 -- TWR for ww synchronization. This method is an incorrect method. TWR requires that a dm-write(x_i) with timestamp TS be ignored if TS is less than the maximum W-timestamp of x_i. This may cause subsequent dm-reads to read inconsistent data; see figure 5.1. (Method 6 is the only incorrect method we will encounter).

Method 7 -- Multi-version T/O for ww synchronization. This method is the correct way to achieve the goals of TWR in conjunction with multi-version rw synchronization. In this method, the pre-commit is accepted iff condition (a) holds. If the pre-commit is accepted, the corresponding dm-write creates a new version of x_i.

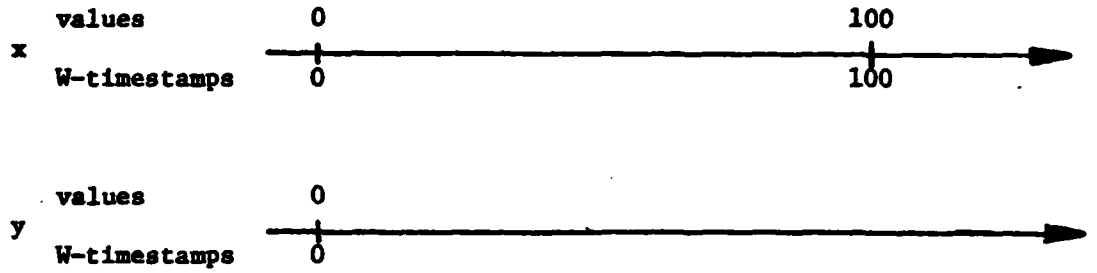
Method 8 -- Conservative T/O for ww synchronization. In this method, a scheduler S will not process a pre-commit with timestamp TS until it has processed all pre-commits with smaller timestamps, and none with larger timestamps. This permits us to simplify the condition for acceptance of a pre-commit as follows. A pre-commit(x_i) with timestamp TS is accepted iff TS is greater than the largest R-timestamp of x_i.

Systematic Forgetting of Old Version

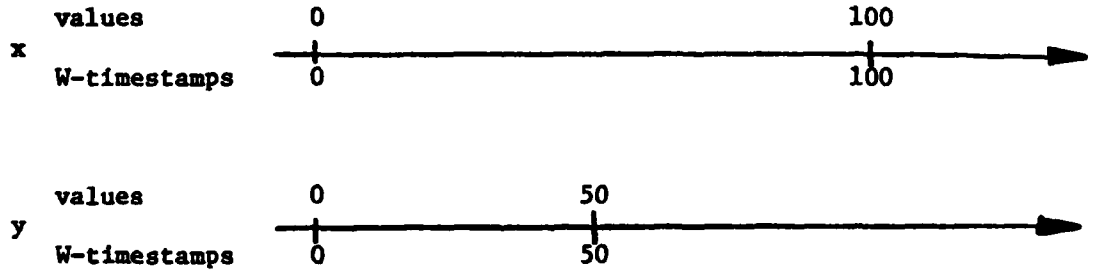
In Methods 5 and 8, the versions of each data item x_i are created in timestamp order. That is, once a version of x_i has been created with timestamp TS, no subsequent transaction can

Inconsistant Retrievals in Method 6 Figure 5.1

- Consider data items x and y with the following versions



- Now suppose T has timestamp 50 and writes x:=50, y:=50. Under Method 6, the update to x is ignored, and the result is



- Finally, suppose T' has timestamp 75 and reads x and y. The values it will read are x=0, y=50 which is incorrect. T' should read x=50, y=50

create a version with a smaller timestamp. Whenever this property holds, it is possible to forget (i.e., delete) old versions in such a way that we never delete a version that is needed by a later transaction.

Let W-max(x_i) be the maximum W-timestamp of x_i, and let W-min be the minimum value of W-max(x_i) over all data items x_i in the database. Observe that no pre-commit with timestamp smaller than W-min can be accepted in the future, since W-min is smaller than W-max(x_i) for all x_i. Consequently, if any update transaction is assigned a timestamp smaller than W-min at any point in the future, the transaction is guaranteed to be restarted. So, insofar as update transactions are concerned, we can safely forget all versions of every data item timestamped less than W-min.

Once old versions are forgotten, any dm-read with timestamp less than W-min must be rejected, since the version it needs no longer exists. It is therefore wise to alert all TMs of the current value of W-min, so they do not issue dm-reads with smaller timestamp. Also, after a new W-min is selected, older versions should not be forgotten immediately, so that active queries timestamped less than the new W-min have an opportunity to finish reading those older versions.

Notice also that Methods 5 and 8 only require that the largest R-timestamp of each data item be stored. Smaller R-timestamps may be forgotten at once.

Systematic Reading of Old Versions

Methods 5 and 8 also support a systematic technique for assigning old timestamps to queries (see Section 4.2.10) in such a way that

a. no dm-read issued by a query will ever invalidate a dm-write; and
b. the timestamp assigned to the query is the largest one satisfying (a).

This technique is similar to the technique for systematic forgetting of old versions.

Let Q be a query. The technique we describe requires that Q's readset be predeclared. Before Q begins its main execution Q's readset is examined; for each x_i in the readset, W-max(x_i) is ascertained. In addition, we calculate W-min = min{W-max(x_i) | x_i is in Q's readset}. The timestamp assigned to Q is W-min - 1.

This technique satisfies properties (a) and (b) by the following argument. Since Methods 5 and 8 create versions in timestamp order, these methods will never accept an update that conflicts with Q but has timestamp less than W-min. Thus, no dm-read issued by Q can ever cause a dm-write to be rejected, as required by property (a). Now, consider any data item x in Q's readset for which W-max(x)= W-min. Methods 5 and 8 could accept

an update transaction T with timestamp W-min+1 that writes into x. If the timestamp of Q were any larger than W-min - 1, Q's dm-read on x would invalidate T. Thus W-min - 1 is the largest timestamp that could be assigned to Q without violating property (a). Q.E.D.

Note that this technique requires more communication than the "non-systematic" algorithm for reading old versions described in Section 4.2.10. It is unclear whether the benefits of the systematic algorithm are worth this extra cost.

5.2.3 Methods Using Conservative T/O for rw Synchronization

The remaining T/O methods use conservative T/O for rw synchronization. In these methods, a scheduler S will not process a dm-read(x_i) with timestamp TS until it has processed all pre-commits with smaller timestamps and no pre-commits with larger timestamps. Symmetrically, S will not process a pre-commit(x_i) with timestamp TS until it has processed all dm-reads with smaller timestamps and none with larger timestamps. When S processes a pre-commit(x_i) with timestamp TS, the action taken depends on the specific ww technique.

Method 9 -- Basic T/O for ww synchronization. The pre-commit is accepted iff TS is greater than the W-timestamp of x_i.

Method 10 -- TWR for ww synchronization. The pre-commit is always accepted. However, if TS is less than the W-timestamp of x_i, the corresponding dm-write has no effect on the database.

Method 10 is essentially the concurrency control of SDD-1 [BSR]. In SDD-1, however, the method is refined in several ways to reduce delay. First, SDD-1 uses classes and conflict graph analysis and requires predeclaration of readsets. In addition, SDD-1 only enforces the conservative scheduling rule on dm-reads. That is, SDD-1 forces dm-reads to wait for pre-commits, but does not insist that pre-commits wait for all dm-reads with smaller timestamps. Consequently, it is possible for dm-reads to be rejected in SDD-1. The SDD-1 designers were willing to accept this possibility for two reasons:

1. Since readsets are predeclared, all dm-reads are issued before the transaction begins its main execution; so the cost of rejecting a dm-read is modest.
2. The probability that a dm-read will be rejected can be reduced by assigning large timestamps to transactions.

Method 11 -- Multi-version T/O for ww synchronization. The pre-commit is always accepted and the corresponding dm-write always creates a new version of x_i. When multi-versions are used, the conservative rw technique can be optimized as follows: a dm-read can never be rejected, and so there is no reason to

force pre-commits to wait for dm-reads. (dm-reads must still wait for pre-commits to ensure that pre-commits are never rejected; and each dm-read must read the appropriate version relative to its timestamp.)

Method 12 -- Conservative T/O for ww synchronization. In this method, scheduler S will not process a pre-commit with timestamp TS until it has processed all pre-commits with smaller timestamps and none with larger timestamps. Combined with conservative rw synchronization, the effect is to process all operations in timestamps order.

Method 12 is a popular timestamp-based concurrency control method. It has been recommended by [KNTH, SM].

5.3 Mixed 2PL and T/O Methods

This section considers concurrency control methods that combine 2PL and T/O techniques. The major difficulty in constructing such methods lies in developing the interface between the 2PL and the T/O technique. Each technique guarantees an acyclic ->rwr (resp. ->ww) relation when used for rw (resp. ww) synchronization. The interface between a 2PL and a T/O technique must guarantee that the combined -> relation -- i.e., ->rwr U ->ww -- remains acyclic. In other words, the interface must ensure that the serialization order induced by the rw

technique is consistent with the serialization order induced by the ww technique.

In Section 5.3.1 we describe an interface that makes this guarantee. Given such an interface, any 2PL technique can be integrated with any T/O technique. Sections 5.3.2 and 5.3.3 describe such methods.

5.3.1 The Interface

The serialization order induced by any 2PL technique is determined by the locked-points of the transactions that have been synchronized (see Section 4.1.1). The serialization order induced by any T/O technique is determined by the timestamps of the synchronized transactions.

One way to interface 2PL and T/O is to use locked-points to induce timestamps. Associated with each data item is a timestamp called an L-timestamp to distinguish it from the R- and W-timestamps needed for T/O synchronization. When a transaction sets a lock on data item x, it simultaneously notes the L-timestamp of x. When the transaction reaches its locked-point it determines the maximum L-timestamp that it noted while setting locks. The timestamp assigned to the transaction is any number greater than that maximum*. When a transaction releases a

lock on x it updates the L-timestamp of x to the maximum of (a) the L-timestamp's current value, or (b) the timestamp of the transaction.

We can prove that timestamps generated in this way are consistent with a serialization order of the transactions that were synchronized by 2PL. Let T_1 and T_n be a pair of transactions such that T_1 must precede T_n in any serialization. This means that there exists a set of transactions $\{T_1,\ldots, T_n\}$ such that (a) the locked-point of T_1 precedes the locked-point of T_2 which precedes the locked-point of $T_3\ldots$, and (b) for each pair of transactions, T_i and T_{i+1} for $i=1,\ldots,n-1$, there exists a data item x_i such that T_i released its lock on x_i before T_{i+1} obtained its lock on x_i. Therefore, the L-timestamp that T_{i+1} noted for x_i is greater-than-or-equal-to the timestamp of T_i and so the timestamp of T_{i+1} is strictly greater-than the timestamp of T_i. It follows that the timestamp of T_n is greater-than the timestamp of T_1. Q.E.D.

---

*Most T/O techniques require that transactions be assigned unique timestamps. Standard strategies can be adopted to meet this requirement. For example, each TM can generate timestamps in the "standard" way described in Section 4.1.1 and the standard timestamp can be appended to low order bits of the timestamp generated here.

5.3.2 Mixed Methods Using 2PL for rw Synchronization

There are twelve principal methods in which 2PL is used for rw synchronization and T/O is used for ww synchronization. These are

#	rw technigue	ww technigue
1	basic 2PL	basic T/O
2	basic 2PL	TWR
3	basic 2PL	multi-version T/O
4	basic 2PL	conservative T/O
5	primary copy 2PL	basic T/O
6	primary copy 2PL	TWR
7	primary copy 2PL	multi-version T/O
8	primary copy 2PL	conservative T/O
9	centralized 2PL	basic T/O
10	centralized 2PL	TWR
11	centralized 2PL	multi-version T/O
12	centralized 2PL	conservative T/O

Method 2 best exemplifies this class of methods, and it is the only one we shall describe in detail.

Method 2 requires that every stored data item have an L-timestamp and a W-timestamp. (It is possible to use a single

timestamp for both roles, but we will not consider this optimization here.)

Let X be a logical data item with copies $x_1, \ldots, x_m$. To read X, a transaction T issues a dm-read on any copy of X, say x_i. This dm-read implicitly requests a read-lock on x_i, and when the read-lock is granted, the L-timestamp of x_i is returned to T. To write into X, T issues pre-commits on every copy of X. These pre-commits implicitly request rw write-locks on the corresponding copies of x_i, and as each write-lock is granted, the corresponding L-timestamp is returned to T. When T has obtained all of its locks, a timestamp for T is calculated as per Section 5.3.1. T attaches this timestamp to its dm-write operations which are then transmitted.

Dm-write operations are processed by DMs in accordance with TWR. Consider a dm-write on x_j with timestamp TS. If TS is greater-than the W-timestamp of x_j, the dm-write is processed as usual -- i.e., the value of x_j is updated to the value contained in the dm-write, and the W-timestamp of x_j is updated to TS. If, however, TS is less-than the W-timestamp of x_j, the dm-write is ignored.

The interesting property of this method is that write-locks never conflict with write-locks. The write-locks obtained by pre-commits are only used for rw synchronization, hence only conflict with read-locks. This method permits transactions to

execute concurrently to completion even if their writesets intersect. Such concurrency is never possible in a pure 2PL method.

Methods 1 and 3-12 are implemented similarly. The integration of two-phase commit into each method requires some care, however.

5.3.3 Mixed Methods Using T/O for rw Synchronization

It is also possible to construct twelve principal methods in which T/O is used for rw synchronization and 2PL is used for ww synchronization.

#	rw technique	ww technique
13	basic T/O	basic 2PL
14	basic T/O	primary copy 2PL
15	basic T/O	voting 2PL
16	basic T/O	centralized 2PL
17	multi-version T/O	basic 2PL
18	multi-version T/O	primary copy 2PL
19	multi-version T/O	voting 2PL
20	multi-version T/O	centralized 2PL
21	conservative T/O	basic 2PL
22	conservative T/O	primary copy 2PL

23	conservative T/O	voting 2PL
24	conservative T/O	centralized 2PL

These methods all require pre-declaration of write-locks. Since T/O is used for rw synchronization, transactions must be assigned timestamps before they issue dm-reads. However, the timestamp generation technique of Section 5.3.1 requires that a transaction be at its locked-point before the transaction is assigned its timestamp. It follows that every transaction must be at its locked-point before it issues any dm-reads, or in other words, every transaction must obtain all of its write-locks before it begins its main execution.

To illustrate how these methods are implemented, we shall describe Method 17. Method 17 requires that each stored data item have a set of R-timestamps, a set of <W-timestamps, value> pairs -- i.e., versions -- and an L-timestamp. Importantly, the L-timestamp of any data item may be taken to be the maximum of its maximum R-timestamp and its maximum W-timestamp.

Let T be a transaction. Before beginning its main execution, T issues pre-commits on every copy of every data item in its writeset. These pre-commits serve three functions: they play a role in ww synchronization, rw synchronization, and the interface between these techniques.

Consider a pre-commit(x_i). The "ww role" of this pre-commit is to request a ww write-lock on x_i. When the ww write-lock is granted, the L-timestamp of x_i is returned to T; this is the "interface role" of the pre-commit. Let TS be the L-timestamp returned at this point. The rw role of the pre-commit is to inform the rw synchronization mechanism that a dm-write(x_i) with timestamp greater-than TS is "pending". The rw technique uses this information to guarantee that the dm-write can be accepted when it ultimately arrives; to make this guarantee the rw technique will not output any dm-read(x_i) with timestamp greater-than TS until the dm-write arrives (see Section 4.2.9).

When T has obtained all of its write-locks, a timestamp for T is calculated as per Section 5.3.1. At this point T begins its main execution.

T attaches its timestamp to all dm-read and dm-write operations that it issues. These operations are processed using the multi-version technique described in Section 4.2.4. A dm-read(x_i) with timestamp TS reads the version of x_i with largest W-timestamp less than TS; this dm-read also adds TS to the set of R-timestamps for x_i. A dm-write(x_i) with timestamp TS creates a new version of x_i with W-timestamp equal to TS.

One interesting property of this method is that restarts are only needed to resolve ww conflicts. In particular, restarts are only needed to prevent or break deadlocks caused by ww

synchronization. rw conflicts never cause restarts. This property cannot be attained by a pure 2PL method. This property can be attained by pure T/O methods, but only if conservative T/O is used for rw synchronization; in many cases conservative T/O introduces excessive delay or is otherwise infeasible.

The behavior of this method for queries is also interesting. Queries, of course, set no write-locks, and so the timestamp generation rule does not apply to queries. This means that the system is free to assign any timestamp it wants to a query. It may assign a small timestamp, in which case the query will read old data but is unlikely to be delayed by "pending" dm-writes. Or it may assign a large timestamp, in which case the query will read current data but is more likely to be delayed. No matter what timestamp is selected, however, a query can never cause an update to be rejected. This property cannot be easily attained by any pure 2PL or T/O method.

We also observe that this method creates versions in timestamp order, and so systematic forgetting of old versions is possible, (see Section 5.2.2.) In addition, the method only requires that the maximum R-timestamp of data items be stored; smaller R-timestamps may be instantly "forgotten".

6. Performance of Concurrency Control Methods

Concurrency control methods are complex algorithms and a quantitative analysis of concurrency control performance is beyond the state-of-the-art. In this section we discuss concurrency control performance in qualitative, comparative terms. We shall compare the performance of principal methods along several dimensions, and we shall discuss the performance impact of refinements to principal methods. Our goal is to provide an intuitive understanding of the factors that influence concurrency control performance.

An overview of the section appears in Section 6.1; the body of our performance analysis follows in Sections 6.2-6.7. Section 6.8 reviews past works on this topic.

We emphasize at the outset that our performance analysis is based on several assumptions and simplifications. Given the present state-of-the-art, these assumptions are necessary in order to say anything at all about concurrency control performance. The reader is cautioned to keep this in mind when interpreting the results in this section.

6.1 Overview

The performance of a concurrency control method is judged in terms of system throughput and user response time. Four cost factors govern the throughput and response time of a concurrency control method: (1) inter-site communication overhead; (2) local processing overhead; (3) number and cost of transaction restarts induced by the method; and (4) the magnitude of transaction blocking caused by the method.

The impact of each cost factor on system throughput and response time varies from system to system and application to application. Consider, for example, a DDBMS whose sites are mainframe computers and whose network is a narrow bandwidth packet-switched network. In this system, communication overhead is expected to have a greater impact on performance than local processing overhead. However, in a DDBMS consisting of micro-computers connected by a wide bandwidth broadcast network, the opposite relationship is likely to hold.

The impact of cost factors on throughput and response time is not understood in detail, however, and we will not focus our analysis on this issue.

Instead, we will concentrate on the performance of principal methods (and important refinements) relative to individual cost factors. For each cost factor we will determine which method has best performance when we consider that cost factor in isolation, which method has worst performance, etc. We summarize the results for each cost factor in diagrams that indicate the relative performance of principal methods under that cost factor. This analysis is carried out in Sections 6.2-6.5.

We will see that no concurrency control method is optimum under all four cost factors; this means that no method is best under all conditions. The relative performance of methods will vary from system to system and application to application, depending on the relative importance of each cost factor for the system and application. Little is known about the magnitude of this variation, nor is it known whether any method offers good performance over a large range of systems and applications.

However, it is possible to identify 11 methods as dominant methods. For any system and application, one of these dominant methods is expected to be optimum. (That is, if the assumptions used in the analysis are valid.) To select a concurrency control method for a given environment, therefore, the system designer need only consider the 11 dominant methods. Dominant methods are discussed in Section 6.6.

The choice among dominant methods requires that the strengths and weaknesses of each method be matched against system and application demands. This remains a largely unsolved engineering problem. Additional research is needed to quantify the performance of dominant methods, to quantify the impact of each cost factor on throughput and response time, and to characterize the important "system and application demands". In Section 6.7, however, we sketch a plausible design scenario for two stereotyped application environments, called optimistic and pessimistic applications. We show that only a handful of methods (no more than 3 or 4) deserve serious attention in these environments. Thus, for these cases at least, the design of a concurrency control appears to be a tractable engineering problem.

6.2 Communication Overhead

There are many components to communication cost in a distributed processing environment. One important component is the total communication volume -- i.e., the total number of bits sent between sites. Another important component is the number of inter-site interactions -- i.e., the total number of messages sent between sites. In addition, distance, network topology, and queuing effects may be important also.

To simplify our analysis we shall make a bold assumption: we will assume that the dominant cost component is number-of-messages, and we will ignore all other factors. Our justification for ignoring communication volume is that most concurrency control messages are short and so the cost of transmitting these messages is almost certainly dominated by "per-message" rather than "per-bit" costs. Our justification for ignoring distance, topology, and queuing effects however is entirely pragmatic: (1) distance and topology effects are difficult to characterize in general-purpose networks; and (2) queuing effects are difficult to characterize without embarking on a quantitative analysis. We would not be surprised to discover that distance, topology, and queuing effects are important in practice. Nonetheless, we ignore these factors here to keep the analysis manageable.

6.2.1 Baseline Communication Requirements

Consider an arbitrary transaction T. A certain amount of communication is needed to process T in a distributed database even if we ignore concurrency control. For every data item that T reads, a dm-read operation must be sent; for every data item that T writes, a pre-commit and a dm-write must be sent. This minimum communication forms a baseline against which concurrency control methods must be judged.

However, this analysis is complicated by an important detail. From a performance standpoint, communication requirements are determined by the number of physical messages transmitted between sites, while the notion of operation as used above is a logical concept. For example, if T writes into several data items at a single DM, the dm-writes for these data items can be packaged as a single physical message. Similarly, if T accesses a data item whose DM is physically co-located with T's TM, no physical messages are needed at all.

Consequently, the baseline communication requirements of a transaction are a function of four parameters: (1) the transaction's readset and writeset; (2) the TM at which the transaction executes; (3) the distribution of data among DMs; and (4) an execution policy that governs when a transaction may issue dm-read and pre-commit operations.

The first three parameters are clear. The fourth parameter covers a spectrum ranging from predeclaration of readsets and writesets to continuous data-item-by-data-item access. In the former case, all dm-reads and pre-commits are sent at one point during transaction execution -- namely immediately before the transaction begins its main execution. This execution policy takes maximum advantage of the ability to package multiple operations into a single physical message. Consequently, the predeclaration policy induces the minimum baseline requirements

of any execution policy. At the other extreme, the continuous policy causes each operation to be transmitted as a separate physical message. Hence, this policy induces the maximum baseline requirements of any execution policy.

The standard execution policy, described in Section 2 lies between these extremes. In the standard policy, dm-reads are issued more or less continuously during transaction execution but all pre-commits are issued when the transaction completes its main execution.

Notice that execution policy can substantially impact the baseline communication requirements of transactions. The choice of execution policy may well have a greater impact on communication cost than the choice of concurrency control method.

6.2.2 Communication Overhead of 2PL Techniques

In this section we examine the communication overhead of 2PL techniques -- i.e., techniques described in Section 4.1. We first consider rw synchronization techniques, then ww techniques, and finally refinements of these techniques.

rw Techniques

Three 2PL techniques are available for rw synchronization: basic 2PL, primary copy 2PL, and centralized 2PL.

When basic 2PL is used for rw synchronization, explicit lock-release operations are needed to release read-locks. These operations introduce communication overhead. However, due to the distinction between operations and physical messages, it is not correct to conclude that the overhead of this technique equals 1 message per data item read. For example, all read-locks held by a transaction at a single DM may be released by a single physical message. Also, if the transaction is writing any data items at the DM, the lock-release operations may be physically packaged with the corresponding dm-write operations. Thus the overhead of basic 2PL for rw synchronization is more accurately characterized as 1 message per DM at which the transaction reads, but does not write. Notice that this overhead is independent of execution policy.

Primary copy 2PL has higher overhead than basic 2PL for rw synchronization, because all transactions must request and release read-locks on primary copies. For example, suppose transaction T wishes to access logical data item X, and suppose a non-primary copy of X is co-located with T's TM. If basic 2PL were used for rw synchronization, T could read that copy of X without sending any physical messages. Under primary copy 2PL, however, T must communicate with the primary copy of X before it

may read its "local" copy. Thus, the communication overhead of primary copy 2PL is monotonically greater than the overhead of basic 2PL for rw synchronization.

Centralized 2PL exhibits a different kind of communication behavior for rw synchronization than either of the preceding techniques. In centralized 2PL locks are requested and released at a centralized 2PL scheduler. The overhead of this technique is directly proportional to the number of distinct points during transaction execution at which locks may be requested. (Notice that read-locks and write-locks must both be obtained at the centralized scheduler for rw synchronization.) In other words, the overhead of centralized 2PL depends upon the execution policy.

Under pre-declaration, all locks are obtained before transaction execution begins.
The overhead under this policy is

1 message to request locks
+1 message to release locks

which equals 2 messages per transaction. Under the standard policy, this overhead increases to a maximum of

1 message per dm-read to request read-locks
+1 message per transaction to request write-locks.
+1 message per transaction to release locks.

Under the continuous policy, the overhead increases further to a maximum of

1 message per dm-read or pre-commit to request locks
+1 message per transaction to release locks.

This qualitative analysis supports the following conclusions:

(i) basic 2PL is cheaper than primary copy 2PL for rw synchronization independent of execution policy;

(ii) basic 2PL is cheaper than centralized 2PL, under the standard and continuous execution policies; and

(iii) centralized 2PL is cheaper than basic 2PL under the predeclaration policy, if the average transaction reads data from more than 2 non-local DMs at which the transaction does not also write.

ww Techniques

The analysis for ww synchronization is simpler. In this case, basic 2PL, primary copy 2PL, and voting 2PL incur no additional overhead over the baseline because write-locks are implicitly requested by pre-commits and are implicitly released by dm-writes. However, centralized 2PL does incur overhead which again depends upon the execution policy. Under the predeclaration and standard policies, the overhead is

1 message to request write-locks

+1 message to release write-locks

which equals 2 messages per transaction. Under the continuous policy this overhead increases to a maximum of

1 message per pre-commit to request write-locks

+1 message per transaction to release write-locks

The overhead of 2PL methods is discussed in Section 6.2.4.

Refinements

Each 2PL technique must incorporate a deadlock resolution technique. A number of such techniques are described in Sections 4.1.6 - 4.1.9. From a communication standpoint, these techniques may be grouped into four categories.

1. Non-preemptive deadlock prevention -- In these techniques, every decision to restart a transaction is made locally, by an individual 2PL scheduler. Thus, no communication overhead is incurred.
2. Preemptive deadlock prevention -- These techniques require the cooperation of a 2PL scheduler and a TM in order to restart a transaction. Thus, some communication overhead is incurred by these techniques. The magnitude of this cost has never been analyzed.
3. Centralized deadlock detection -- This technique requires that each 2PL scheduler transmit "waits-for"

information to a central site on a periodic basis. This periodic transmission constitutes communication overhead. Notice, however, that if centralized 2PL is used, then centralized deadlock detection is free.

4. Hierarchical deadlock detection -- In these techniques local deadlocks are detected without any communication. However, periodic transmission of waits-for information is needed to detect multi-site deadlocks. An interesting special case is two-level hierarchies: the bottom level of the hierarchy consists of 2PL schedulers, while the top level is essentially a centralized deadlock detector. The effect is to reduce the communication overhead of centralized deadlock detection, by avoiding the need to transmit local waits-for information.

Notice also that two-level hierarchies are optimal given our simplifying assumptions regarding communication cost. This is a case where distance and network topology effects play a critical role in determining the communication cost of a technique.

Each 2PL technique may also be refined by applying the heuristics of Section 4.1.10. Of these heuristics, though, only predeclaration has an impact on communication requirements; this impact has already been considered.

6.2.3 Communication Overhead of T/O Techniques

In principle, T/O techniques incur no additional communication overhead beyond the baseline requirements. That is, to read a data item using a T/O technique, a transaction need only issue a dm-read on that data item; no lock-releases or other operations are needed. And to write a data item using a T/O technique, the only necessary operations are a pre-commit and a dm-write. Thus, no communication is required beyond the baseline requirements.

In practice, however, the conservative T/O technique is likely to require substantial overhead in the form of null and request-null messages in order to avoid intolerable delays. This overhead has unusual load-dependent properties. Recall that TMs are required to periodically send null operations to TMs when the TMs have no "real" operations to send. This condition will arise most often when there is little load. As load increases, however, more and more TMs will have real operations to send to DMs, thereby decreasing the need for null operations. Consequently the overhead of conservative techniques tends to decrease as load increases. A quantitative analysis supporting this observation is reported by [KNTH]. By

contrast, the communication overhead of 2PL methods monotonically increases with load.

Refinements

Several refinements are available for reducing the communication overhead of conservative T/O techniques. Classes and conflict graph analysis tend to reduce communication by reducing the number of TMs that must communicate with each DM on a regular basis. However, these techniques can also increase communication because they force transactions to execute at designated TMs; if a transaction is submitted at an "incorrect" TM, the transaction must be transmitted to a TM where it is permitted to execute. The infinite timestamp technique of Section 4.2.5 can also be used to reduce the overhead of conservative T/O methods.

Predeclaration also impacts the communication requirements of T/O methods, by impacting the baseline requirements for transaction execution. This effect has already been analyzed in Section 6.2.1.

6.2.4 Communication Overhead of Principal Methods

Figure 6.1 summarizes the relative communication overhead of synchronization techniques.

Diagrams of the type used in figure 6.1 will appear throughout our analysis. Intuitively, the higher a technique is in the diagram, the greater its cost -- i.e., techniques at the top of the diagram have worst performance and techniques at the bottom have best performance. Formally, we draw an edge from Technique A to Technique B if our analysis leads us to conclude that A has higher cost than B.

Returning to figure 6.1, notice that the relative performance of techniques depends upon operating conditions in two respects.

1. The performance of centralized 2PL is better than noncentralized 2PL if and only if the following condition holds:

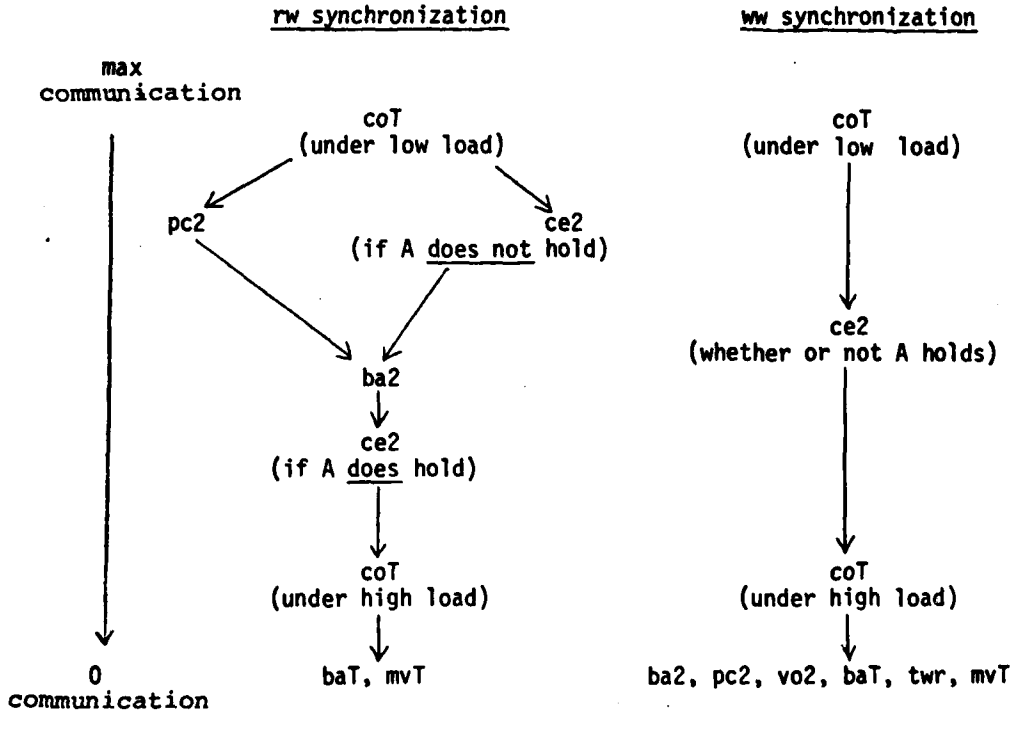
   Condition A -- predeclaration is in effect and the average transaction reads data from more than two DMs at which the transaction does not also write.

2. The overhead of conservative T/O is very high under conditions of low load, but tends toward zero as load increases.

Communication Overhead
of 2PL and T/O Techniques

Figure 6.1

Legend: (i) ba2 = basic 2PL; pc2 = primary copy 2PL;
vo2 = voting 2PL; ce2 = centralized 2PL
(ii) baT = basic T/O; twr = Thomas Write Rule;
mvT = multi-version T/O; coT = conservative T/O



The relative overhead of principal methods can be approximated from figure 6.1, since the overhead of a principal method is approximately equal to the overhead of its rw technique plus its ww technique. For some methods, however, the overhead is below this estimate because of synergistic effects. In particular, if centralized 2PL is used for all synchronization, and if predeclaration is in effect, the messages needed for ww synchronization can be physically packaged with the rw messages. Similarly, the overhead of using conservative T/O is approximately the same whether it is used for rw synchronization, ww synchronization, or both. On the other hand, the overhead of mixed methods tends to exceed the rw plus ww overhead because of difficulties in integrating two-phase commit. (See Section 5.3.)

Notice that not every pair of techniques has costs that are comparable in figure 6.1. Two nodes of a diagram are called incomparable if there is no path from one to the other. For example, pc2 and ce2 (if condition A does not hold) are incomparable in figure 6.1 for rw synchronization. This means that primary copy 2PL will have more overhead than centralized 2PL in some situations but not others. We remark that future quantitative analysis might eliminate incomparabilities of this nature. For example, quantitative analysis might indicate that in all practical situations, one or the other technique is always better.

Figure 6.2 summarizes the relative communication overhead of 2PL and T/O methods. The relative overhead of mixed methods follows a similar pattern; a comprehensive comparison of mixed methods will not be attempted, however, because detailed implementations of these methods have not been worked out.

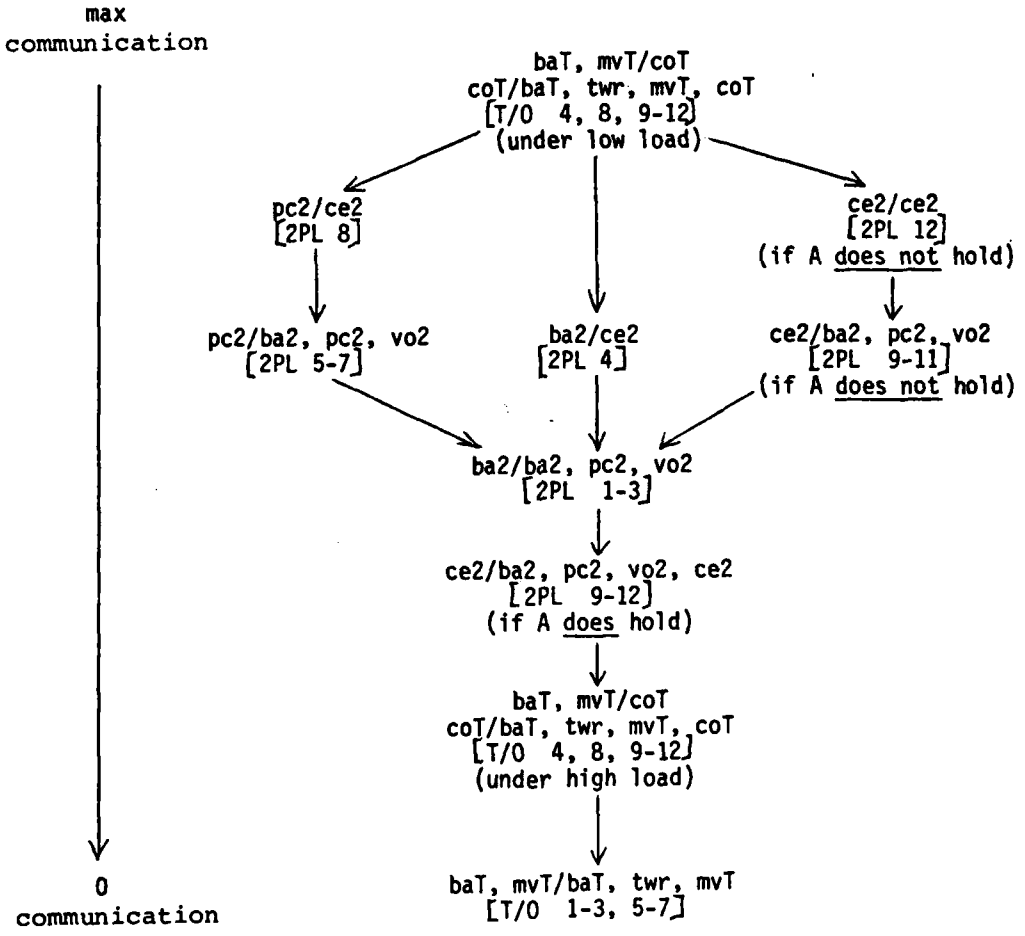
The preceding analysis overlooks an optimization that is possible when primary copy 2PL is used for rw synchronization with primary copy or voting 2PL for ww synchronization (i.e., 2PL Methods 6 and 7). Under this optimized policy, transactions may issue pre-commits continuously during their execution. However, these pre-commits are not sent to all copies of the data items being updated. For Method 6, these pre-commits are only sent to primary copies; for Method 7, these pre-commits are sent to a majority of copies. Pre-commits are sent to all other copies when the transaction terminates. Thus, this policy is a compromise between the standard and continuous execution policies.

This new policy offers better performance than the continuous policy for Methods 6 and 7. In the absence of quantitative data, however, we are unable to compare the performance of Methods 6 and 7 under the new policy to the performance of other methods under the usual policies.

Communication Overhead
of 2PL and T/O Methods

Figure 6.2

Legend: (i), (ii) see figure 6.1
(iii) the notation "x/y" means use technique x for rw synchronization and technique y for ww synchronization
(iv) [2PL number] = 2PL method number (see Section 5.1)
[T/O number] = T/O method number (see Section 5.2)



6.3 Local Processing Overhead

Every technique described in Section 4 places some demand on local computation resources. In the absence of quantitative performance data it is difficult to estimate the magnitude of these demands. However, it is possible to compare the local processing overhead of principal methods in general terms.

We believe that the major source of local processing overhead for principal methods is the maintenance and utilization of synchronization information. By "synchronization information" we mean locks, timestamps, and multi-versions. Numerous data structures can be imagined for the storage of this information, and these data structures can be accessed by many different algorithms. Because of these myriad differences, it is impractical to estimate local processing cost in any absolute sense. However, qualitative analysis is possible if we make a few reasonable assumptions.

First, we assume that the cost of maintaining and utilizing synchronization information is an increasing function of its volume. For example, if Method A requires more locks than Method B, then Method A will have more local processing overhead than Method B. This assumption is clearly reasonable.

Second, we assume that the unit cost of maintaining and utilizing lock information is identical to the unit cost of timestamps. For example, if Method A requires N locks and Method B requires N timestamps, the methods are assumed to have identical cost. This assumption is less clear-cut than the first assumption. However, we believe it to be reasonable because the same basic operations must be supported relative to both kinds of synchronization: given the name of a data item, say x, we must be able to retrieve and update the synchronization information for x. Any algorithm that implements these operations for lock information can be adapted to handle timestamps for essentially the same cost, and vice versa.

Finally, we assume that the unit cost of extra versions (for multi-version T/O) is greater-than the unit cost of locks or timestamps. This assumption is reasonable because an extra version consists of a timestamp and a value.

Given these assumptions we analyze the local processing overhead of 2PL techniques and methods in Section 6.3.1; we analyze T/O techniques and methods in Section 6.3.2; and we compare 2PL methods to T/O methods in Section 6.3.3. Mixed methods are considered briefly in Section 6.3.4.

6.3.1 Local Processing Overhead of 2PL Techniques and Methods

Each 2PL technique requires lock information for all data items being accessed by active transactions. Once a transaction terminates, all of its locking information may be forgotten.

The precise lock requirements differ from technique to technique because of differences in the way redundant data is handled. Consider a logical data item X with copies $x_1, \ldots, x_n$, and suppose transaction T wishes to update X. If basic 2PL is used for rw or ww synchronization, write-locks must be obtained on all copies of X. However, if primary copy 2PL is used for rw and ww synchronization, a write-lock need only be set on one copy of X, namely the primary copy. Centralized 2PL has the same requirements for lock information as primary copy 2PL. Voting 2PL lies between basic 2PL and primary copy 2PL; under voting 2PL, write-locks must be set on a majority of copies of X.

The overhead of principal 2PL methods can be inferred from the above.

(i) All 2PL methods require read-locks for data items that are being read by active transactions.

(ii) All 2PL methods that use basic 2PL for rw or ww synchronization (i.e., Methods 1-4, 5, 9) require write-locks on all copies of logical data items being updated by active transactions.

(iii) 2PL methods that use primary copy or centralized 2PL for rw and ww synchronization (i.e. Methods 6, 8, 10, 12) require a single write-lock for each logical data item being updated by active transactions.

(iv) 2PL methods that use primary copy or centralized 2PL for rw synchronization and voting 2PL for ww synchronization (i.e., Methods 7, 11) require write-locks on a majority of copies of each logical data item being updated by active transactions.

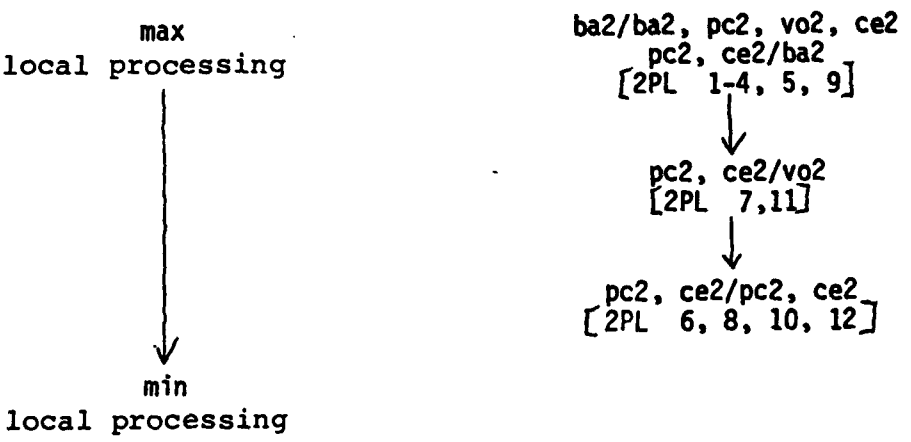
This analysis is summarized in figure 6.3.

6.3.2 Local Processing Overhead of T/O Techniques and Methods

Each non-conservative T/O technique requires timestamps for data items accessed by recent transactions, since timestamps are only forgotten after a pre-determined time interval has elapsed. Multi-version techniques require that extra versions be maintained for recently accessed data items. Conservative T/O techniques do not require any timestamps for data items -- these techniques only require timestamped operations.

Local Processing Overhead
of 2PL and T/O Methods

Figure 6.3



The local processing overhead of T/O methods is as follows.

(i) T/O Methods 1, 2, and 4 (basic T/O for rw synchronization and basic T/O, TWR, and conservative T/O for ww) require R-timestamps for all data items read by recent transactions and W-timestamps for all data items written by recent transactions.

(ii) T/O Method 3 (basic T/O for rw and multi-version T/O for ww) requires an R-timestamp for each items read by a recent transaction, and extra versions for each data item written by a recent transaction.

(iii) T/O Methods 5 and 8 (multi-version T/O for rw synchronization, with basic T/O and conservative T/O for ww synchronization) have the same requirements as Method 3

(iv) T/O Method 7 (multi-version T/O for rw and ww) requires multiple R-timestamps (resp. extra versions) for each data item read (resp. written) by a recent transaction.

(v) T/O Methods 9 and 10 (conservative T/O for rw, basic T/O and TWR for ww) require a W-timestamp for each data item written by a recent transaction. These methods do not require any R-timestamps.

(vi) T/O Method 11 (conservative T/O for rw synchronization and multi-version T/O for WW) requires extra versions for each data item written by a recent transaction.

(vii) T/O Method 12 (conservative 2PL T/O for all synchronization) requires no data timestamps at all.

Figure 6.4 summarizes the local processing overhead of T/O methods.

6.3.3 Comparison of 2PL and T/O Methods

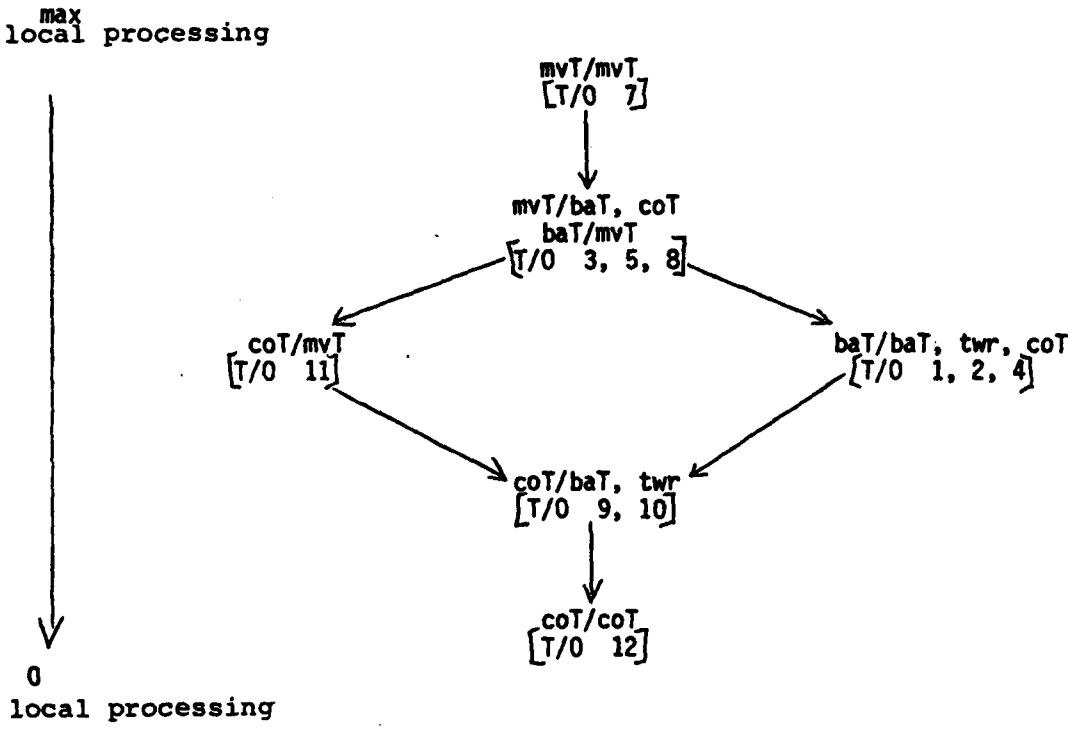
We may compare the local processing overhead of 2PL and T/O methods if we assume that every active transaction is also considered to be a recent transaction. This amounts to assuming that timestamps of active transactions are never forgotten, which is probably a reasonable assumption.

Given this assumption, we can compare the local processing overhead of the basic 2PL Method (2PL Method 1) to that of the basic T/O method (T/O Method 1). The 2PL method requires read- and write-locks for data accessed by active transactions, while the T/O method requires R- and W-timestamps for data accessed by recent transactions. Since the set of active transactions is a subset of the recent transactions (by assumption), the number of locks needed by the 2PL method is less-than-or-equal -to the number of timestamps needed by the T/O method. This translates into lower cost for the 2PL method.

Since 2PL Method 1 is the most expensive 2PL method relative to local processing overhead (see figure 6.3), it follows that all 2PL methods have lower processing cost than T/O Method 1.

Local Processing Overhead
of 2PL Methods

Figure 6.4



However, all 2PL methods have greater processing overhead than T/O Method 12 (conservative T/O for all synchronization), since the latter method requires no stored synchronization information.

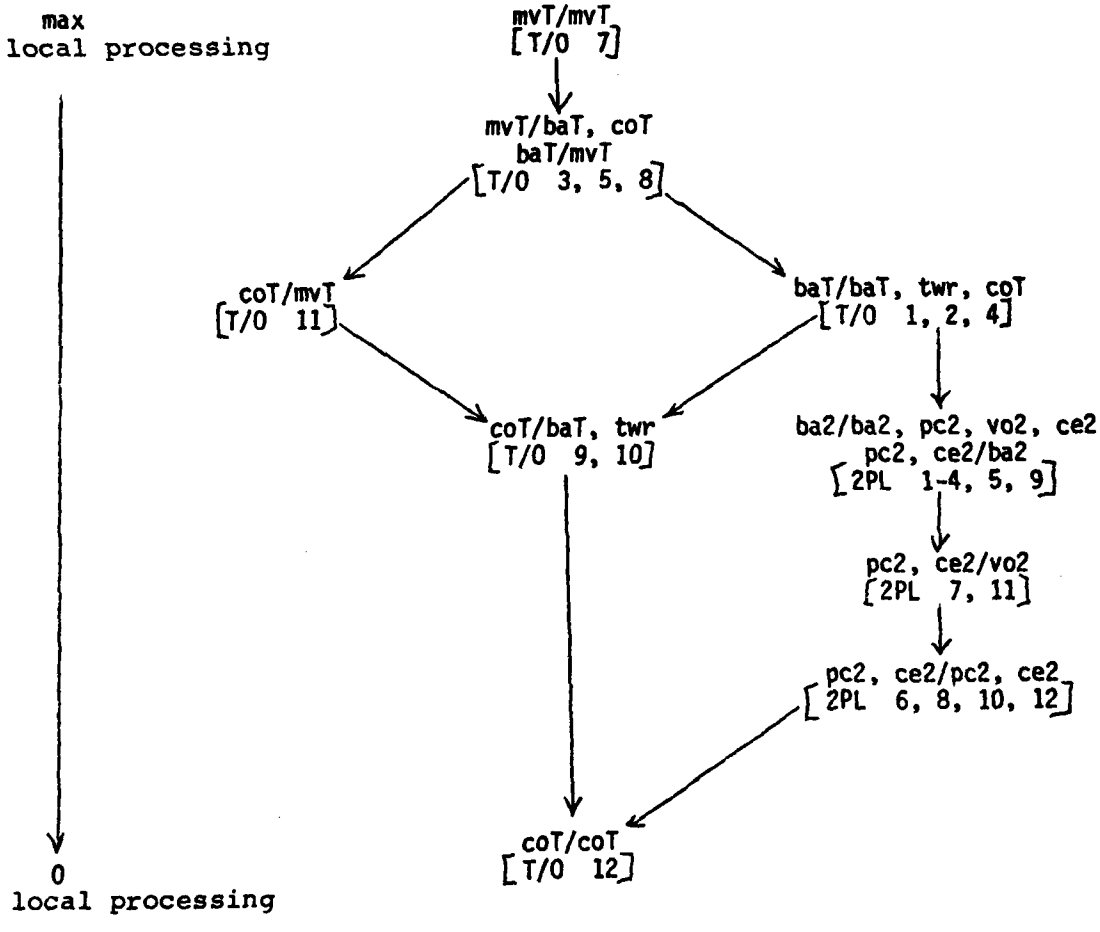
This analysis is summarized in figure 6.5

6.3.4 Local Processing Overhead of Mixed Methods

All mixed methods require data item timestamps or multi-versions in order to generate transaction timestamps using the algorithm of Section 5.3.1. Mixed methods that use multi-version T/O require approximately as much synchronization information as T/O Methods 3, 5, and 8; other mixed methods have requirements similar to T/O Methods 1, 2, and 4.

Local Processing Overhead
of T/O Methods

Figure 6.5



6.4 Transaction Restarts

Most concurrency control methods have the ability to restart transactions either for deadlock resolution purposes or as a normal part of their operation. The number of restarts induced by a method is an important component of the method's cost. If the number of restarts grows too large the system can thrash, wasting much of its effort executing transactions that are subsequently restarted. The cost per restart is also an important performance factor.

6.4.1 Restart Behavior of 2PL Techniques

The restart behavior of 2PL techniques depends largely on the choice of deadlock resolution techniques. The best restart behavior is exhibited by deadlock detection techniques (assuming the phantom deadlock problem can be controlled; see Section 4.1.8). With these techniques transactions are only restarted when a deadlock actually occurs. In addition, these techniques are able to select inexpensive "victim" transactions to restart. By contrast, deadlock prevention techniques induce restarts when the danger of deadlock exists, and these techniques are not at

liberty to select "victims" on the basis of restart cost. Thus, deadlock detection induces fewer restarts and has a lower cost per restart than deadlock prevention.

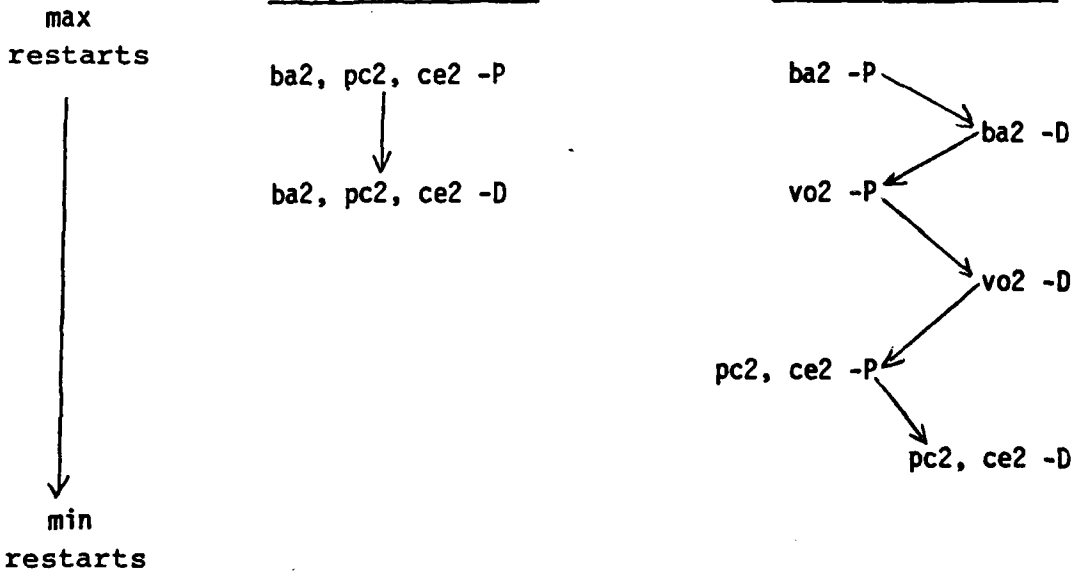
For ww synchronization, there are additional differences among 2PL techniques because of differences in the way redundant data is handled. Consider a logical data item X with copies $x_1,...x_n$ and suppose two concurrent transactions wish to write into X. If basic 2PL is used for ww synchronization, these transactions can deadlock simply due to the order in which they obtain locks on $x_1,...x_n$. For example, if one transaction locks x_1 first and the other transaction locks x_2 first, the result is deadlock. However, if primary copy 2PL, voting 2PL, or centralized 2PL are used for ww synchronization, deadlocks of this type cannot arise.

We can further distinguish between these techniques by observing that primary copy 2PL and centralized 2PL are immune to this type of deadlock no matter how many concurrent transactions try to update X. Voting 2PL, however, is only immune when two or fewer transactions are involved. For example, if three transactions simultaneously try to update X, each could obtain write-locks on one-third of the copies, thereby preventing any transaction from obtaining a majority.

The restart behavior of 2PL techniques is summarized in figure 6.6.

Restart Behavior of 2PL Techniques Figure 6.6

Legend: "-D" means deadlock detection;
"-P" means deadlock prevention



Predeclaration also has a substantial impact on restart behavior. Predeclaration forces run-time conflicts to occur before transactions begin their main execution. This tends to reduce the cost per restart -- if a transaction is restarted during its predeclaration phase there is less wasted effort than if the transaction is restarted near the end of its execution.

However, predeclaration also tends to increase the number of restarts for two reasons. First, it is often difficult to predict the data item that a transaction will access before it executes. To be on the safe side, it is necessary to lock all data items the transaction might access. These extra locks increase the probability of run-time conflict and deadlock. And second, predeclaration causes locks to be held for longer periods of time, which also increases the probability of run-time conflict and deadlock.

More analysis is needed to determine whether (or under what conditions) the good tendencies of predeclaration outweigh the bad.

6.4.2 Restart Behavior of T/O Techniques

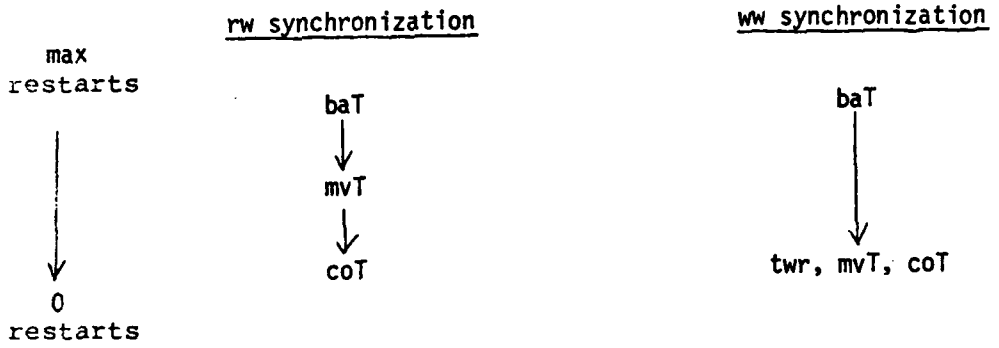
The restart behavior of T/O techniques varies over a wide range. Optimal restart behavior is exhibited by conservative T/O, the Thomas Write Rule, and multi-version T/O (for ww synchronization), because these techniques never induce restarts. The worst restart behavior is exhibited by basic T/O: this technique induces a restart whenever a DM receives conflicting operations out of timestamp order. Multi-version T/O (for rw synchronization) lies between these extremes.

The restart behavior of T/O techniques is summarized in figure 6.7.

Predeclaration reduces the cost per restart for T/O techniques just as it does for 2PL techniques.

Predeclaration may also reduce the number of restarts when non-conservative T/O techniques are used for rw synchronization. This effect may occur if (a) clocks in different TMs are reasonably well-synchronized, and (b) the network delivers messages with approximately equal delay. If these conditions are met, predeclaration will tend to cause DMs to receive operations in timestamp order, thereby reducing the likelihood

Restart Behavior of T/O Techniques Figure 6.7



of rejection. This effect has been reported in simulation studies by [Ries 1].

Predeclaration may also increase the number of restarts for non-conservative T/O techniques because of the need to "pre-access" all data items a transaction might access.

6.4.3 Comparison of 2PL and T/O Techniques

In this section we compare the restart behavior of 2PL techniques to that of T/O techniques.

We begin by comparing basic-2PL-with-deadlock-prevention to basic T/O. To do so, we adopt the Wait-Die technique of [RLS] as our "standard" deadlock prevention technique, see Section 4.1.7. The Wait-Die technique induces a restart whenever a DM receives conflicting operations for active transactions out of timestamp order. Basic T/O, on the other hand, induces a restart whenever a DM receives conflicting messages for any transactions out of timestamp order. Thus we see that Wait-Die has similar, but somewhat better, restart behavior than basic T/O.

Notice also that basic-2PL-with-deadlock-prevention (i.e. Wait-Die) has the worst restart behavior of any 2PL technique (see figure 6.6). Since basic T/O is worse than Wait-Die, it

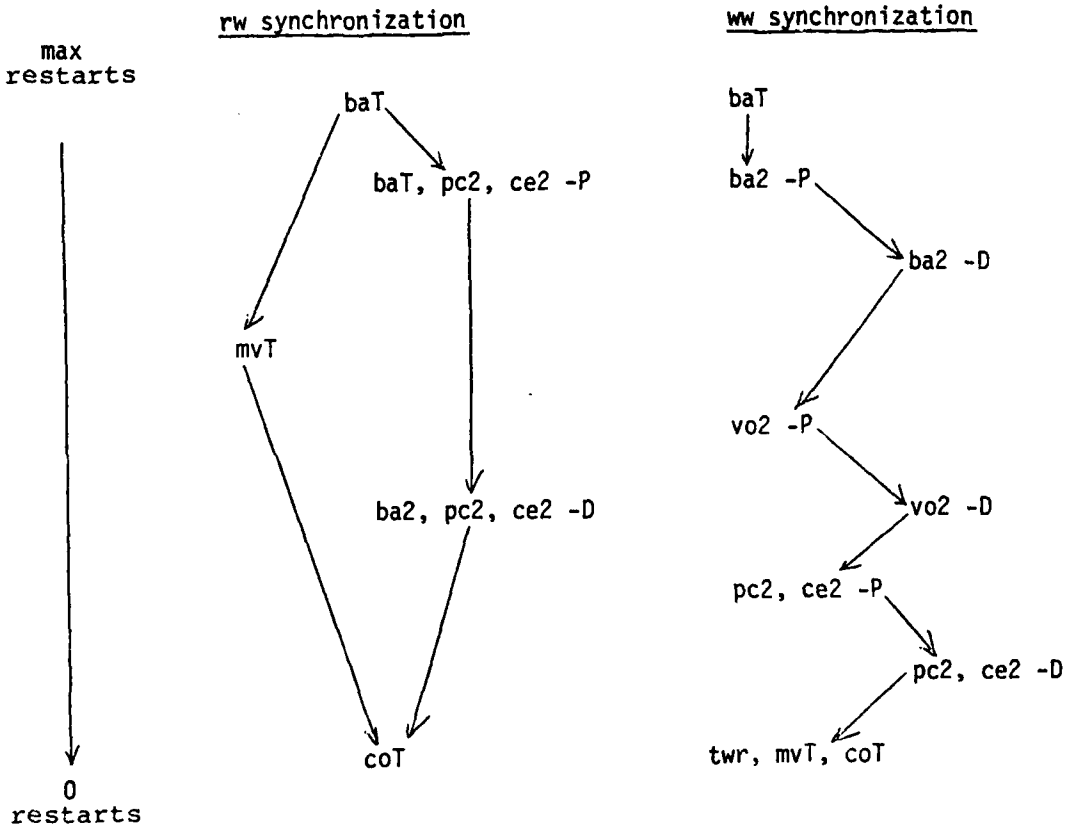
follows that basic T/O has worse restart behavior than any 2PL technique.

At the opposite end of the restart spectrum, we have techniques which induce no restarts: conservative T/O, the Thomas Write Rule, and multi-version T/O (for ww synchronization). Since these techniques induce no restarts, they have better restart behavior than any 2PL technique.

Insofar as ww synchronization is concerned, this completes the comparison of 2PL and T/O techniques. See figure 6.8.

For rw synchronization, however, we must compare multi-version T/O to the various 2PL techniques. Consider multi-version T/O vs. basic-2PL-with-deadlock-prevention (i.e. Wait-Die). In one way, multi-version T/O is better: under multi-version T/O dm-reads never induce restarts, whereas under Wait-Die a "late" dm-read can sometimes cause a restart. In particular, under Wait-Die, a dm-read(x) with timestamp TS will induce a restart if some transaction with a larger timestamp is holding a write-lock on x. On the other hand, there are situations in which Wait-Die is better than multi-version T/O. For example, under Wait-Die a pre-commit(x) with timestamp TS can safely arrive after a dm-read(x) with larger timestamp, provided the transaction issuing the dm-read had already terminated.* Under multi-version T/O, however, the pre-commit would be rejected.**

Restart Behavior of 2PL and T/O Techniques Figure 6.8



We see no qualitative way of deciding which of the above effects is most likely. In the absence of quantitative analysis, we cannot decide whether multi-version T/O or Wait-Die has better restart behavior.

It is impossible to directly compare multi-version T/O to any other 2PL techniques for similar reasons.

Figure 6.8 summarizes the relative restart behavior of 2PL and T/O techniques.

6.4.4 Restart Behavior of Principal Methods

As far as we know, the restart behavior of a method can be inferred from the restart behavior of its rw and ww techniques. We know of no synergistic effects that complicate this analysis. Thus, the relative restart behavior of principal methods can be inferred from figure 6.8. This behavior is summarized in figure 6.9.

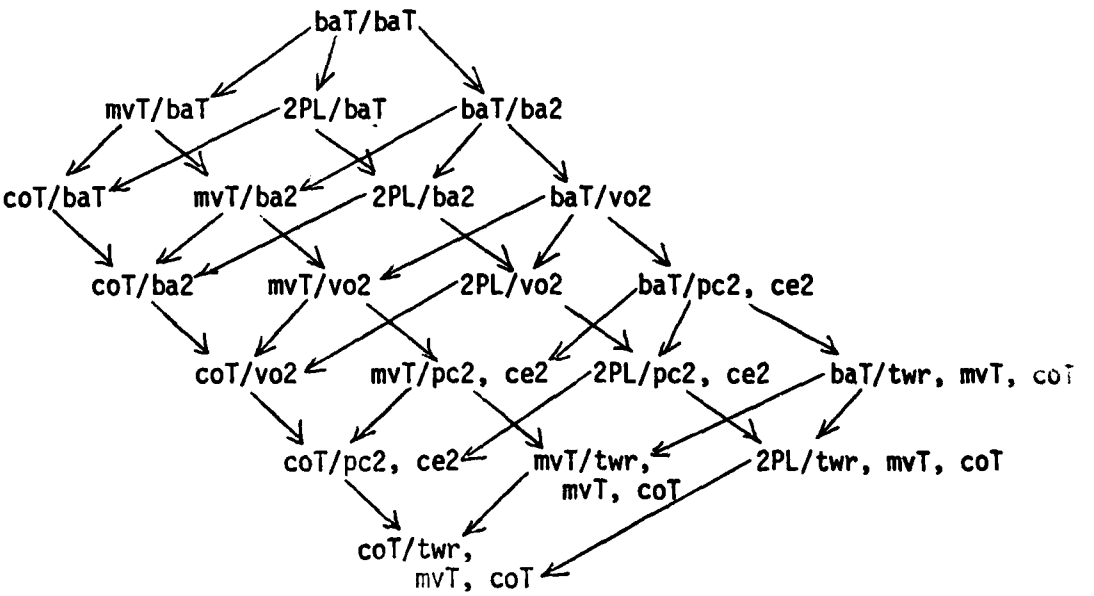
Notice that figure 6.9 includes 2PL, T/O, and mixed methods.

---

*Technically it is only necessary for the transaction to have released its read-lock on x.
**Unless another pre-commit(x) with timestamp TS", TS<TS"<TS had already been received.

Restart Behavior of Principal Methods Figure 6.9

Legend: (i) 2PL = any 2PL technique
(ii) any 2PL technique can use deadlock prevention or detection; in all cases prevention induces more restarts than detection



6.5 Transaction Blocking

Many synchronization techniques have the ability to block transactions -- i.e., to suspend or intentionally delay their execution. For example, 2PL techniques will block a transaction if it requests an unavailable lock. Blocking is a complementary tactic to transaction restart. Both tactics are used to prevent run-time conflicts from causing non-serializable operation. Blocking, however, is a much cheaper tactic than restarts -- a block merely delays a transaction whereas a restart forces the transaction to be re-executed. The number of blocks induced by a concurrency control method and the delay per block are important components of the method's cost.

6.5.1 Blocking Behavior of 2PL Techniques

The blocking behavior of 2PL techniques depends principally on the deadlock resolution scheme. Deadlock detection induces more blocking than deadlock prevention. Under deadlock detection a transaction T will be blocked if and only if T requests a lock that conflicts with a lock owned by another transaction T'. T will remain blocked until (a) T' releases its lock, or (b) T or

T' is restarted by the deadlock detector. Under deadlock prevention, however, the system does not always block T when it requests a conflicting lock. Instead, the system can restart T or the transaction with which it conflicts. Thus deadlock prevention induces a smaller number of blocks than deadlock detection. In addition, the delay per block is smaller since no deadlocks are possible.

Notice that the blocking behavior of these techniques is exactly complementary to their restart behavior.

Predeclaration also influences blocking behavior. Predeclaration tends to increase the number of blocks because of the need to lock all data items a transaction might access. Predeclaration also tends to increase the delay per block since locks are held for longer periods of time.

6.5.2 Blocking Behavior of T/O Techniques

The blocking behavior of T/O techniques varies widely. At one extreme we have conservative T/O. Tnis technique uses blocking as its normal mode of operation to guarantee that conflicting operations are processed in timestamp order. The price paid for this conservatism is that transactions are blocked even when no conflicts exist.

At the other extreme we have the non-conservative T/O techniques. These techniques never induce blocking when used for ww synchronization. When used for rw synchronization, these techniques have intermediate blocking behavior. Basic T/O is required to block dm-read operations under the following conditions: a DM will block a dm-read(x) with timestamp TS if it has received a pre-commit(x) with timestamp less-than TS, but has not yet received the corresponding dm-write(x); this blocking is necessary for two-phase commit to work properly as described in Section 4.2.9. Multi-version T/O is required to block dm-read operations under slightly less restrictive conditions also described in Section 4.2.9. Neither technique is ever required to block pre-commit or dm-write operations.

Various refinements also influence the blocking behavior of T/O techniques. Classes, conflict graph analysis, and the "infinite timestamp" heuristic of Section 4.2.5 improve the blocking behavior of conservative T/O. The "delay" heuristic of Section 4.2.10 degrades the blocking behavior of non-conservative T/O techniques in order to improve restart behavior.

6.5.3 Comparison of 2PL and T/O Techniques

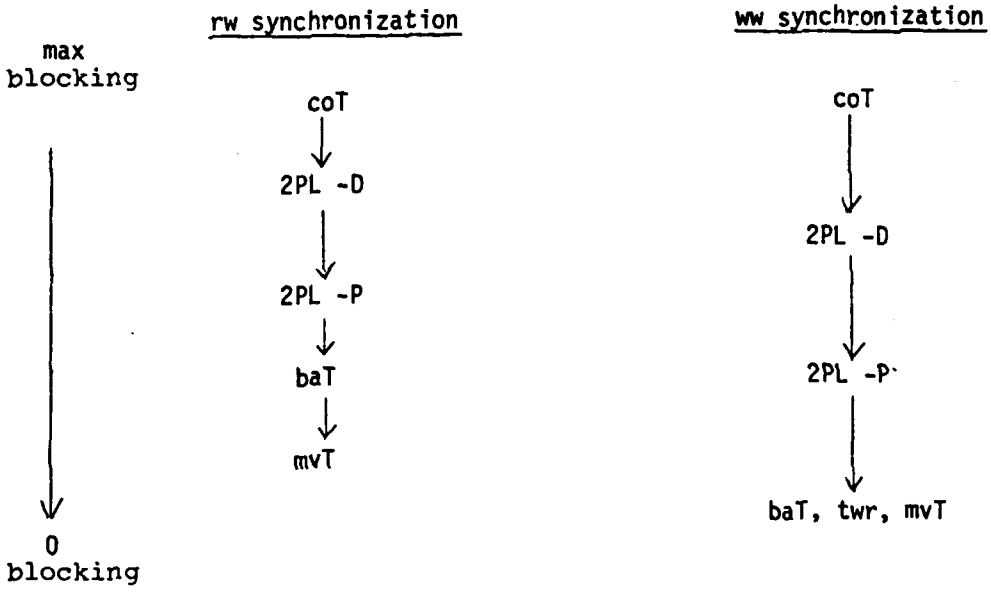
We now compare the blocking behavior of 2PL and T/O techniques.

For ww synchronization the comparison is straightforward. Conservative T/O exhibits the worst behavior, since conservative T/O blocks transactions even in the absence of conflicts. Non-conservative T/O exhibits the best behavior, since these techniques never block transactions. 2PL techniques lie between these extremes, since these techniques block transactions when run-time conflicts are deleted: of course, deadlock prevention has better blocking behavior than deadlock detection. This comparison is summarized in figure 6.10.

For rw synchronization the analysis is more complex. Let us begin by comparing basic 2PL with deadlock prevention to basic T/O. As in Section 6.4, we adopt Wait-Die as our standard deadlock prevention technique for purposes of this comparison.

Wait-Die is required to block dm-read operations under the following conditions: a DM will block a dm-read(x) with timestamp TS if a transaction with smaller timestamp owns a write-lock on x; this latter condition arises if the DM has received a pre-commit(x) with timestamp less-than TS, but has

Blocking Behavior of 2PL and T/O Techniques Figure 6.10



not yet received the corresponding dm-write. Basic T/O is required to block dm-read messages under the exact same conditions; see Section 6.5.2. Thus, Wait-Die and basic T/O have identical blocking insofar as dm-read messages are concerned.

These techniques have different blocking behavior, however, when pre-commits are considered. Wait-Die is required to block a pre-commit(x) with timestamp TS if a transaction with smaller timestamp owns a read-lock on x. Basic T/O is never required to block pre-commit operations. Thus Wait-Die has worse behavior than basic T/O when we consider pre-commit operations.

Since Wait-Die and basic T/O have identical blocking behavior for dm-reads, while Wait-Die has worse behavior for pre-commits, we conclude that Wait-Die has worse blocking behavior overall.

The following observations will conclude the comparison for rw synchronization.

(i) The blocking behavior of 2PL techniques depends principally on the deadlock resolution technigue, i.e., primary copy 2PL and centralized 2PL have similar behavior to basic 2PL when all use the Wait-Die deadlock prevention technique. Consequently, all 2PL techniques that use Wait-Die have worse blocking behavior than basic T/O.

(ii) All 2PL techniques with deadlock detection have worse behavior than 2PL techniques with deadlock prevention, see Section 6.5.1. With deadlock detection every run-time conflict causes a transaction to be blocked, but transactions are not blocked in the absence of conflict.

(iii) Conservative T/O has worse blocking behavior than any 2PL techniques, since conservative T/O blocks transactions even when conflicts do not exist.

(iv) Multi-version T/O has the best blocking behavior of any technique, since this technique is even better than basic T/O; see Section 6.5.2.

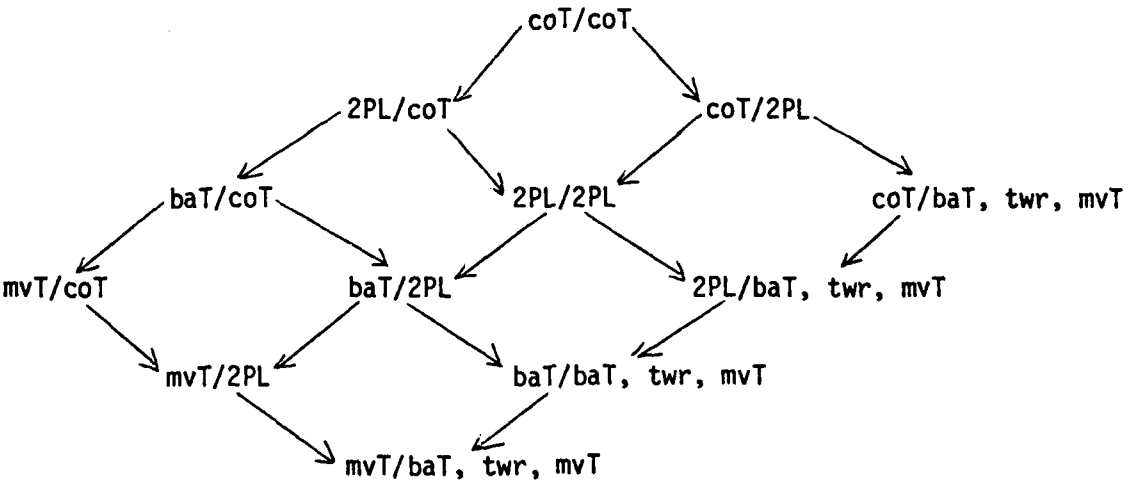
This comparison is summarized in figure 6.10.

6.5.4 Blocking Behavior of Principal Methods

As in the case of restart behavior, it appears that the blocking behavior of a method can be inferred from the blocking behavior of its component techniques. Figure 6.11 summarizes this behavior.

Blocking Behavior of Principal Methods Figure 6.11

Legend: any 2PL technique can use deadlock detection or deadlock prevention. In all cases, prevention has better blocking behavior than detection.



6.6 Dominant Methods

Our analysis of concurrency control performance is summarized in figures 6.2, 6.5, 6.9, and 6.11. The reader can observe that no concurrency control method is optimal under all cost factors. 2PL methods tend to rank in the middle of the field under all cost factors. Non-conservative T/O methods have minimum communication overhead and good blocking behavior, but moderate local processing overhead and poor restart behavior. Conservative T/O methods vary even further: they have minimum local processing overhead and optimal restart behavior, but very poor blocking behavior; their communication overhead is high under low load, but low under high load. Thus no method has best performance under all conditions.

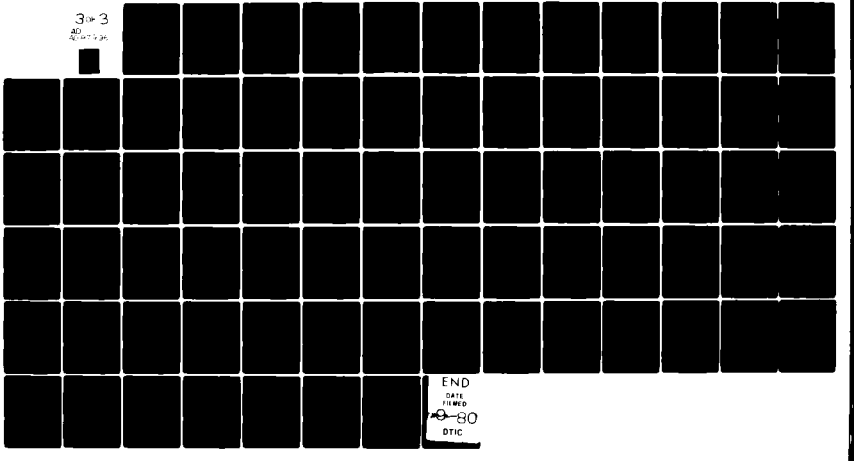
However, we can identify some methods which are better than certain other methods under all conditions. We say that Method A dominates Method B if A is better than B under all cost factors; Method A is a dominant method if no other method dominates it. For any given system and application, some dominant method will have optimal performance. When selecting a concurrency control method for a given environment, one need only consider dominant methods; all other methods are irrelevant.

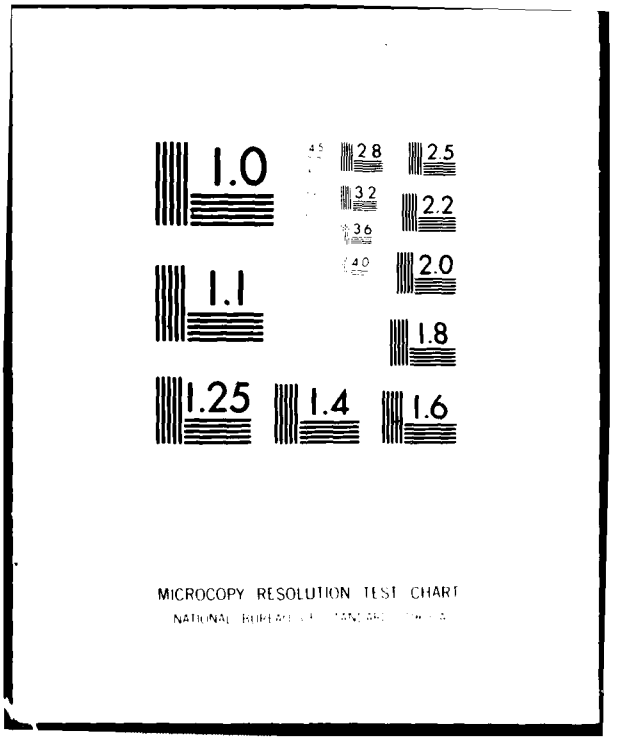
In Section 6.6.1 we examine the relative performance of principal 2PL methods under all four cost factors, and we show that four 2PL methods are dominant. They are 2PL Method 2 (basic 2PL for rw synchronization, with primary copy 2PL for ww); 2PL Method 6 (primary copy 2PL for all synchronization); 2PL Method 10 (centralized 2PL for rw synchronization, with primary copy 2PL for ww); and 2PL Method 12 (centralized 2PL for all synchronization). For any given system and application, one of these four 2PL methods will out-perform all other 2PL methods; thus the other 8 principal 2PL methods need never be considered when designing a concurrency control method. Noti that the "standard" distributed 2PL method -- 2PL Method 1 (basic 2PL for all synchronization) -- is not one of the dominant methods.

In Section 6.6.2 we examine principal T/O methods in a similar fashion, and again identify four dominant methods. These are T/O Method 2 (basic T/O for rw synchronization, with the Thomas Write Rule for ww); T/O Method 7 (multi-version T/O for all synchronization); T/O Method 10 (conservative T/O for rw synchronization, with the Thomas Write Rule for ww); and T/O Method 12 (conservative T/O for all synchronization).

In Section 6.6.3 we consider mixed methods. Our analysis of mixed methods in Section 5 is not detailed enough to identify all dominant T/O methods. Nonetheless, we identify three

AD-A087 996 COMPUTER CORP OF AMERICA CAMBRIDGE MA F/G 9/2
FUNDAMENTAL ALGORITHMS FOR CONCURRENCY CONTROL IN DISTRIBUTED D--ETC(U)
MAY 80 P A BERNSTEIN, N GOODMAN F30602-79-C-0191
UNCLASSIFIED RADC-TR-80-158 NL




1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6
MICROCOPY RESOLUTION TEST CHART

dominant mixed methods with interesting performance characteristics. They are Mixed Method 2 (basic 2PL for rw synchronization, with the Thomas Write Rule for ww). Mixed Method 6 (primary copy 2PL for rw synchronization, with the Thomas Write Rule for ww); and Mixed Method 10 (centralized 2PL for rw synchronization, with the Thomas Write Rule for ww). In Section 6.7 we will see that these mixed methods out-perform dominant 2PL and T/O methods in certain system and application environments.

Interestingly, 7 of the 11 dominant methods identified in this section have not been previously described in the literature. These are 2PL Methods 2 and 10, T/O Methods 2 and 7, and all three mixed methods. The remaining 4 methods have been described in the following contexts.

(i) 2PL Method 6 is essentially the primary copy algorithm proposed by [Stonebraker]; this is also the algorithm used in the distributed INGRES system [SN].

(ii) 2PL Method 12 is the primary site algorithm proposed by [AD].

(iii) T/O Method 10 is approximately the SDD-1 concurrency control mechanism described by [BSR].

(iv) T/O Method 12 is the algorithm proposed by [SM] for the National Software Works; a similar algorithm is described by [KNTH].

6.6.1 Dominant 2PL Methods

In this section we compare the performance of principal 2PL methods to determine which are dominant. This analysis is organized pedagogically in terms of rw synchronization. We first consider methods that use basic 2PL for rw synchronization; next we consider methods that use primary copy 2PL for rw synchronization; finally, we consider methods that use centralized 2PL for rw synchronization.

Figure 6.12 summarizes the performance of 2PL methods that use basic 2PL for rw synchronization, (figure 6.12 is obtained from figures 6.2, 6.5, 6.9, and 6.11 by eliminating all other methods from the figures). All methods that use basic 2PL for rw synchronization have identical local processing overhead, because all require read-locks and write-locks on all stored data items being accessed by active transaction. In addition, these methods have identical blocking behavior, although the blocking behavior is dependent on the choice of deadlock resolution scheme. Therefore, to choose among these methods, we need only consider communication overhead and restart behavior.

Relative Performance of 2PL Methods
Using Basic 2PL for rw Synchronization
Figure 6.12

Communication Overhead	Local Processing Overhead	Restart Behavior	Blocking Behavior
ba2/ce2 [2PL 4] ↓ ba2/ba2, pc2, vo2 [2PL 1-3]	ba2/2PL [2PL 1-4]	ba2/ba2 [2PL 1] ↓ ba2/vo2 [2PL 3] ↓ ba2/pc2, ce2 [2PL 2,4]	ba2/2PL [2PL 1-4]

Communication overhead for these methods is minimized by selecting any ww technique except centralized 2PL; centralized 2PL incurs extra overhead in these methods because it requires extra messages to request and release write-locks at a central site. Restart behavior is optimized by selecting primary copy 2PL or centralized 2PL for ww synchronization, because these techniques avoid deadlocks caused by ww synchronization on redundant copies of data. Thus, primary copy 2PL is an optimal ww synchronization technique under all cost factors. This means that 2PL Method 2 (basic 2PL for rw synchronization, with primary copy 2PL for ww synchronization) dominates all 2PL methods that use basic 2PL for rw synchronization.

Next, consider methods that use primary copy 2PL for rw synchronization. The performance of these methods under all cost factors is summarized in figure 6.13. We see from the figure that communication overhead is minimized by any ww technique except centralized 2PL. Local processing overhead and restart behavior are both optimized by selecting primary copy 2PL or centralized for ww synchronization. Blocking behavior is not affected by the choice of ww technique. Thus, we see again that primary copy 2PL is an optimal ww technique under all cost factors, and so 2PL Method 6 (primary copy 2PL for all synchronization) dominates all 2PL methods that use primary copy 2PL for rw synchronization.

Relative Performance of 2PL Methods
Using Primary Copy 2PL
for rw Synchronization

Figure 6.13

Communication Overhead	Local Processing Overhead	Restart Behavior	Blocking Behavior
pc2/ce2 [2PL 8] ↓ pc2/ba2, pc2, vo2 [2PL 5-7]	pc2/ba2 [2PL 5] ↓ pc2/vo2 [2PL 7] ↓ pc2/pc2, ce2 [2PL 6, 8]	pc2/ba2 [2PL 5] ↓ pc2/vo2 [2PL 7] ↓ pc2/pc2, ce2 [2PL 6, 8]	pc2/2PL [2PL 5-8]

Finally, consider methods that use centralized 2PL for rw synchronization. The performance of these methods under all cost factors is indicated in figure 6.14. The performance of these methods depends on an application characteristic, namely, condition A of Section 6.2.2. If condition A does not hold, the best ww technique is again primary copy 2PL. However, if condition A does hold, then primary copy 2PL and centralized 2PL are equally effective ww techniques. In this case, it is probably best to select centralized 2PL for ww synchronization, since the resulting method supports centralized deadlock detection at no extra cost. Thus, in practice, there are two dominant methods that use centralized 2PL for rw synchronization: Method 10 (primary copy 2PL for ww), if condition A does not hold; and Method 12 (centralized 2PL for ww), if condition A does hold.

The performance of the four dominant 2PL methods is compared in figure 6.15. All four methods have identical restart and blocking behavior. The choice between these methods is strictly a tradeoff between communication and local processing overhead. Moreover, if condition A holds, 2PL Method 12 (centralized 2PL for all synchronization) simultaneously minimizes both cost factors — i.e., if condition A holds, Method 12 dominates all other 2PL methods. However, if condition A does not hold, one is forced to balance the extra communication overhead of 2PL Methods 6 and 10 against the extra local processing overhead of

Relative Performance of 2PL Methods
Using Centralized 2PL
for rw Synchronization

Figure 6.14

Note: Condition A as defined in Section 6.2.2

Communication Overhead	Local Processing Overhead	Restart Behavior	Blocking Behavior
ce2/ce2 [2PL 12] (if A does not hold) ↓ ce2/ba2, pc2, vo2 [2PL 9-11] (if A does not hold) ↓ ce2/2PL [2PL 9-11] (if A does hold)	ce2/ba2 [2PL 9] ↓ ce2/vo2 [2PL 11] ↓ ce2/pc2, ce2 [2PL 10, 12]	ce2/ba2 [2PL 9] ↓ ce2/vo2 [2PL 11] ↓ ce2/pc2, ce2 [2PL 10, 12]	ce2/2PL [2PL 9-12]

Relative Performance of
Dominant 2PL Methods

Figure 6.15

Communication Overhead	Local Processing Overhead	Restart Behavior	Blocking Behavior
ce2/pc2 [2PL 10] (if A does not hold) ↓ pc2/pc2 [2PL 6] ↓ ba2/pc2 [2PL 2] ↓ ce2/ce2 [2PL 12] (if A does hold)	ba2/pc2 [2PL 2] ↓ pc2/pc2 ce2/pc2, ce2 [2PL 6, 10, 12]	ba2/pc2 pc2/pc2 ce2/pc2, ce2 [2PL 2, 6, 10, 12]	ba2/pc2 pc2/pc2 ce2/pc2, ce2 [2PL 2, 6, 10, 12]

2PL Method 2. This issue is discussed further in Section 6.7.

6.6.2 Dominant T/O Method

We now analyze dominant T/O methods in a similar fashion. Figures 6.16-6.18 summarize the relative performance of T/O methods that use basic T/O, multi-version T/O, and conservative T/O respectively for rw synchronization.

Consider T/O methods that use basic T/O for rw synchronization. Communication overhead and blocking are optimized by selecting any non-conservative technique for ww synchronization. Restarts are minimized (in fact eliminated) by using any ww technique except basic T/O. Computation overhead is minimized by any ww technique except multi-version T/O. Thus, the Thomas Write Rule is optimal under all cost factors, and Method 2 (basic T/O for rw synchronization, TWR for ww) dominates all methods that use basic T/O for rw synchronization.

Next, consider methods that use multi-version T/O for rw synchronization. No method in this class dominates the others; see figure 6.17. Method 7, however, (multi-version T/O for all synchronization) is optimal under all cost factors except computation overhead. As a practical matter, the extra computation overhead of Method 7 over Methods 5 and 8 is probably not significant. All methods require multiple versions

Relative Performance of T/O Methods
Using Basic T/O for rw Synchronization Figure 6.16

Communication Overhead	Local Processing Overhead	Restart Behavior	Blocking Behavior
baT/coT [T/O 4] ↓ baT/baT, twr, mvT [T/O 1-3]	baT/mvT [T/O 3] ↓ baT/baT, twr, coT [T/O 1, 2, 4]	baT/baT [T/O 1] ↓ baT/twr, mvT, coT [T/O 2-4]	baT/coT [T/O 4] ↓ baT/baT, twr, mvT [T/O 1-3]

Relative Performance of T/O Methods
Using Multi-Version T/O
for rw Synchronization

Figure 6.17

Communication Overhead	Local Processing Overhead	Restart Behavior	Blocking Behavior
mvT/coT [T/O 8] ↓ mvT/baT, mvT [T/O 5, 7]	mvT/mvT [T/O 7] ↓ mvT/baT, coT [T/O 5, 8]	mvT/baT [T/O 5] ↓ mvT/mvT, coT [T/O 7, 8]	mvT/coT [T/O 8] ↓ mvT/baT, mvT [T/O 5, 7]

of data items that have been recently updated. The difference is that Method 7 requires multiple R-timestamps for data items that have been recently read, while Methods 5 and 8 only require a single R-timestamp for those data items. It is reasonable to assume that if one can afford the overhead of multiple versions, then one can also afford multiple R-timestamps. Thus, for practical purposes it is reasonable to claim that Method 7 is best over all methods that use multi-version T/O for rw synchronization.

Finally, consider methods that use conservative T/O for rw synchronization. Again, no single method is dominant; see figure 6.18. However, Method 10 (conservative T/O for rw synchronization, TWR for ww) dominates all methods except Method 12 (conservative T/O for all synchronization). Consequently, we identify both of these methods as dominant.

The performance of the dominant T/O methods is compared in figure 6.19. Unlike the dominant 2PL methods, the performance of dominant T/O methods varies widely over all cost factors. This reflects the fundamental difference in "synchronization philosophy" between conservative and non-conservative techniques: conservative T/O is designed to avoid restarts at all cost, while non-conservative techniques are willing to accept restarts as a a normal mode of operation. The wide variation of dominant T/O methods is an important source of

Relative Performance of T/O Methods
Using Conservative T/O
for rw Synchronization

Figure 6.18

Communication Overhead	Local Processing Overhead	Restart Behavior	Blocking Behavior
coT/baT, twr mvT, coT [T/O 9-12]	coT/mvT [T/O 11] ↓ coT/baT, twr [T/O 9, 10] ↓ coT/coT [T/O 12]	coT/baT [T/O 9] ↓ coT/twr, mvT, coT [T/O 10-12]	coT/coT [T/O 12] ↓ coT/baT, twr, mvT [T/O 9-11]

Relative Performance
of Dominant T/O Methods

Figure 6.19

Communication Overhead	Local Processing Overhead	Restart Behavior	Blocking Behavior
coT/twr, coT [T/O 10, 12]	mvT/mvT [T/O 7]	baT/twr [T/O 2]	coT/coT [T/O 12]
↓	↓	↓	↓
baT/twr mvT/mvT [T/O 2, 7]	baT/twr [T/O 2]	mvT/mvT [T/O 7]	coT/twr [T/O 10]
	↓	↓	↓
	coT/twr [T/O 10]	coT/twr, coT [T/O 10, 12]	baT/twr [T/O 2]
	↓		↓
	coT/coT [T/O 12]		mvT/mvT [T/O 7]

flexibility in the selection of concurrency control methods. We return to this point in Section 6.7.

6.6.3 Dominant Mixed Methods

The description of mixed methods in Section 5 is not detailed enough to support analyses similar to Sections 6.6.1 and 6.6.2. However, we shall describe three mixed methods which out-perform dominant 2PL and T/O methods in certain system contexts. These are Methods 2, 6, and 10 of Sections 5.3.2.

Mixed Method 2 uses basic 2PL for rw synchronization and TWR for ww synchronization. This method has the same communication requirements as 2PL Method 2 (basic 2PL for rw, primary copy 2PL for ww), but better restart and blocking behavior, because TWR never induces restarts or blocking. On the negative side, the mixed method has higher local processing overhead; its local processing overhead is comparable to T/O methods that use basic T/O for rw synchronization (e.g. T/O Method 2). One can think of Mixed Method 2 as a compromise between 2PL Method 2 and T/O Method 2.

Mixed Method 6 uses primary copy 2PL for rw synchronization and TWR for ww synchronization. This method has the same communication requirements as 2PL Method 6 (primary copy 2PL for

all synchronization), but better restart and blocking behavior. Mixed Method 6 has more local processing overhead than 2PL Method 6, but less overhead than Mixed Method 2 (because it only requires R-timestamp on primary copies of logical data items).

Mixed Method 10 uses centralized 2PL for rw synchronization and TWR for ww synchronization. This method has the same communication requirements as 2PL Methods 10 and 12 (centralized 2PL for rw, primary copy 2PL or centralized 2PL for ww), but better restart and blocking behavior. Its local processing overhead is similar to Mixed Method 6.

By using the Thomas Write Rule for ww synchronization, each mixed method attains better restart and blocking behavior than the "corresponding" 2PL method. This improvement in restart and blocking behavior is paid for by an increase in local processing overhead. On the other hand, by using 2PL for rw synchronization, the mixed methods attain better restart behavior than T/O Method 2 (basic T/O for rw synchronization, TWR for ww); this improvement is paid for by increased communication and blocking costs. Similarly, the mixed methods have better blocking behavior than T/O Method 10 (conservative T/O for rw, TWR for ww synchronization) which is primarily paid for by increased restarts.

6.7 Designing a Concurrency Control Method

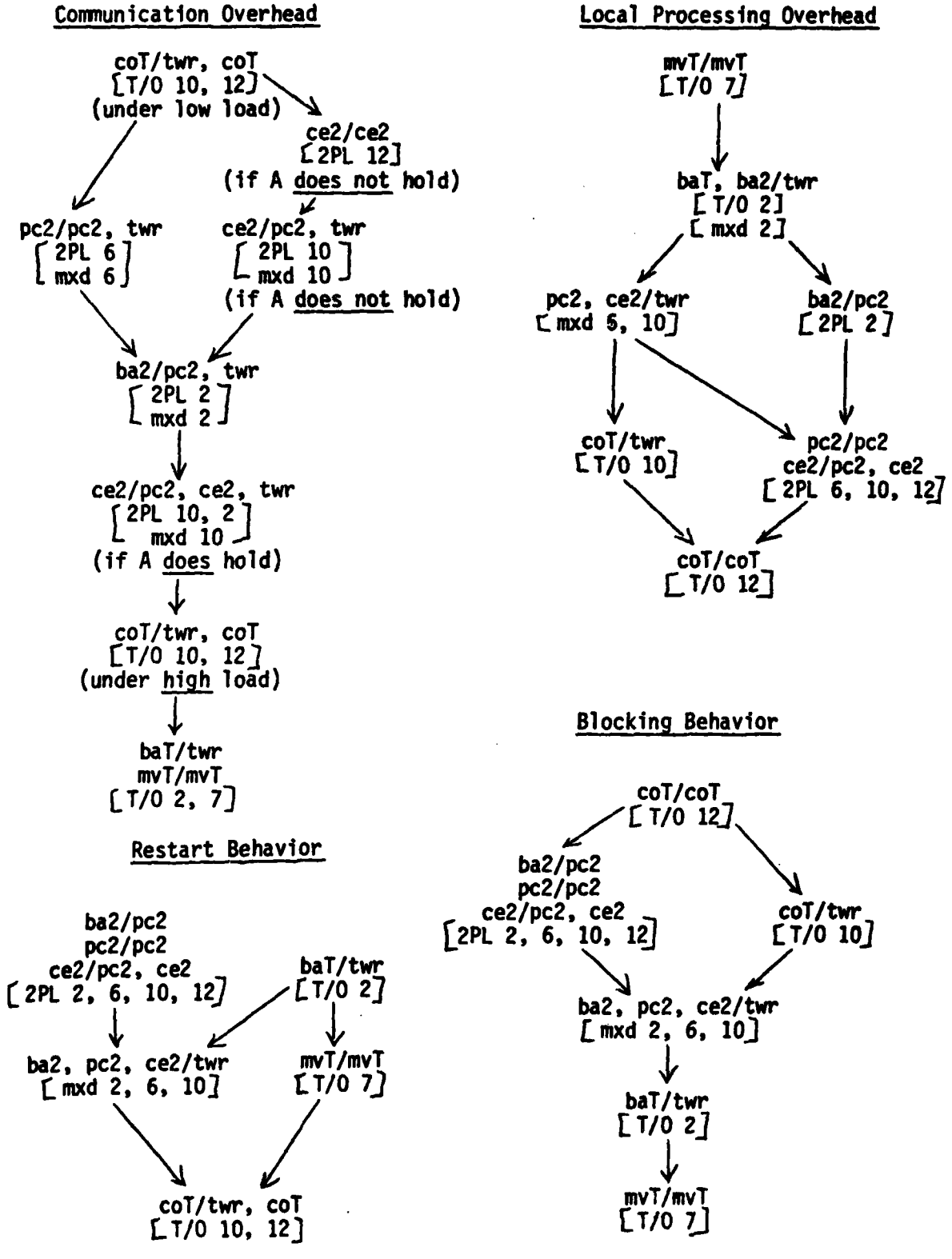
Section 6.6 has identified eleven dominant concurrency control methods. The relative performance of these methods is summarized in figure 6.20. Designing a concurrency control method for a given systems and application amounts to selecting a dominant method whose performance is well-matched to the application's demands. In addition, the dominant method may be fine-tuned by incorporating refinements.

In this section we discuss the impact of system and application factors on concurrency control performance. We present a plausible design scenario for the selection of a concurrency control method in certain stereotypical environments.

We focus on the issue of run-time conflicts. A run-time conflict occurs when two or more concurrent transactions access the same data item in conflicting modes. Run-time conflicts endanger the correct operation of the system, because they can cause non-serializable behavior. To safeguard system operation, all concurrency control methods must incur some cost to ensure that run-time conflicts are synchronized properly.

The severity of the concurrency control problem for an application is largely determined by the rate of run-time

Relative Performance of Dominant Methods Figure 6.20



conflicts. If the rate of run-time conflicts is high, concurrency control performance will inevitably be poor; following [KR] we call this a pessimistic situation. Conversely, if the rate of run-time conflicts is low, concurrency control performance can be quite good; we call this an optimistic situation.

The first step in our design scenario is to determine whether the application is optimistic, pessimistic, or in between. This step will not be discussed in this report. Sections 6.7.1 and 6.7.2 discuss the remainder of the design problem for pessimistic and optimistic applications respectively. The "in-between" situation is considered briefly in Section 6.7.3.

6.7.1 Handling Pessimism

In a pessimistic situation one should expect many run-time conflicts to occur. It is important, therefore, to keep the cost of handling each conflict as low as possible. Transaction restart is generally the most expensive way of responding to a run-time conflict, and so the primary goal in a pessimistic situation is to minimize the number of restarts and the cost per restart. Other cost factors should be considered secondarily.

From figure 6.20 we see that optimal restart behavior is exhibited by T/O Methods 10 and 12. These methods use conservative T/O for rw synchronization and the Thomas Write Rule or conservative T/O for ww synchronization. These methods induce no restarts. On the negative side, these methods have bad blocking behavior and high communication overhead (under low load). These negative aspects can be mitigated to some extent by classes and conflict graph analysis. Also, since load is likely to be high in pessimistic applications, the communication overhead of these methods may be moderate anyway.

The choice between T/O Methods 10 and 12 involves a trade-off between local processing overhead and blocking. Method 10 has higher local processing overhead because it requires W-timestamps for recently updated data items; Method 12 has higher blocking because it requires that dm-writes be processed in timestamp order. In a pessimistic application, the computation overhead of Method 10 is almost certainly worth what it costs and we conclude that Method 10 is preferable to Method 12 in these situations.

For some applications the blocking and communication overhead of Method 10 may be intolerable even if classes and conflict graph analysis are used. For these applications, the designer should consider T/O Method 7 or Mixed Methods 2, 6, and 10. See figure 6.20.

T/O Method 7 uses multi-version T/O for all synchronization. This method induces a restart if a DM receives a pre-commit with timestamp TS after a conflicting dm-read with larger timestamp TS'. In a pessimistic situation, one expects to receive many conflicting pre-commits and dm-reads from concurrent transactions. These conflicting messages will tend to have nearly equal timestamps (assuming clocks at different TMs are reasonably well-synchronized). To avoid excessive restarts we should "stack the deck" by transmitting pre-commits as early as possible and dm-reads as late as possible. In other words, for Method 7 to work well in a pessimistic application we should predeclare writesets but not readsets. As a fringe benefit, predeclaration will also tend to reduce the cost per restart.

Mixed Method 2, 6, and 10 use various 2PL techniques for rw synchronization and the Thomas Write Rule for ww synchronization. Because 2PL is used, a deadlock resolution technique must also be specified. The choice of deadlock resolution technique strongly impacts the restart behavior of the method. To minimize the number of restarts and cost per restart deadlock detection should be used instead of prevention.

Mixed Methods 2 and 6 use non-centralized 2PL techniques for rw synchronization. For these methods, deadlock detection incurs communication overhead since waits-for information must be transmitted to a centralized deadlock detector periodically. A

possibly more damaging drawback is that the deadlock detection algorithm cannot be executed continuously -- since waits-for information is only received periodically, the algorithm can only be run periodically. On average, a deadlock will lie undetected for one-half the interval between executions of the deadlock detector. During this interval, all transactions involved in the deadlock are blocked. This phenomenon may seriously degrade concurrency control performance since many deadlocks are likely to arise in pessimistic situations.

To reduce the blocking cost of deadlock detection, waits-for information may be transmitted more frequently. However, this increases communication overhead.

Mixed Method 10 uses centralized 2PL for rw synchronization. For this method, deadlock detection incurs no extra communication overhead. Also, the deadlock detector may be executed continuously at low cost.

As a practical matter we believe that deadlock detection overhead will prove excessive in pessimistic applications for non-centralized 2PL techniques. In our opinion, the only viable 2PL technique for pessimistic applications is centralized 2PL. To use 2PL for a pessimistic application, the only choices are Mixed Method 10 and 2PL Method 12 (centralized 2PL for all synchronization). Notice that if these methods are selected, predeclaration of readsets and writesets is necessary to avoid excessive communication cost.

The choice between Mixed Method 10 and 2PL Method 12 is a tradeoff between the extra computation overhead of the mixed method vs. its increased concurrency. The mixed method requires R- and W-timestamps in addition to read- and write-locks , but permits concurrent transactions to update the same data items without restarts or blocking. In pessimistic situations, the overhead of timestamps is probably worth what it costs. We conclude that Mixed Method 10 is probably better than 2PL Method 12 in those situations.

The analysis of this section may be summarized as follows.

(i) T/O Method 10 (conservative T/O for rw synchronization, TWR for ww) is probably the best method for pessimistic applications. Classes and possibly conflict graph analysis are useful refinements.

(ii) If the blocking and communication overhead of T/O Method 10 is not acceptable, there are two feasible alternatives: T/O Method 7 (multi-version T/O for all synchronization) and Mixed Method 10 (centralized 2PL for rw with TWR for ww synchronization).

(iii) Both alternatives require predeclaration. T/O Method 7 requires predeclaration of writesets but not readsets; Mixed Method 10 requires predeclaration of readsets and writesets. Predeclaration is only feasible if the database system can accurately estimate the data requirements of transactions before they execute.

(iv) If T/O Method 7 and Mixed Method 10 are both feasible, the choice between them depends on two additional factors both of which entail quantitative analysis: (1) the relative restart behavior of the methods; and (2) the tradeoff between the extra communication overhead of Mixed Method 10 vs. the extra local processing overhead of T/O Method 7.

6.7.2 Handling Optimism

Concurrency control is much simpler for optimistic applications, because few conflicts are expected.

All synchronization techniques only induce restarts in response to conflicts, and all techniques except conservative T/O only induce blocking in response to conflicts. Since few conflicts are expected in optimistic situations, all techniques except conservative T/O can be expected to exhibit good restart and blocking behavior. Conservative T/O induces blocking as a normal mode of operation whether or not conflicts are present. For this reason, we deem conservative T/O to be infeasible in optimistic situations. The relative performance of all other techniques depends only on communication and local processing overhead.

The feasible T/O methods are Method 2 (basic T/O with TWR) and Method 7 (multi-version T/O for all synchronization). Both of these methods have no communication overhead, but Method 2 has less computation overhead; see figure 6.20. This means that T/O Method 2 dominates T/O Method 7 for optimistic applications.

All 2PL methods (i.e., Methods 2, 6, 10, 12) are feasible for optimistic applications. The choice among these methods depends principally on two factors: (1) whether predeclaration is in effect; and (2) the relative importance of communication vs. computation.

If predeclaration is not in effect, centralized 2PL incurs excessive communication overhead. This eliminates Methods 10 and 12 from consideration. Method 2 (basic 2PL for rw, primary copy 2PL for ww) has more computation overhead than Method 7 (primary copy 2PL for all synchronization), since Method 2 is forced to set write-locks on all copies of redundant data. On the other hand, Method 2 has less communication overhead than Method 7, because Method 7 is forced to read primary copies of data items. The choice between these methods is not clear-cut and will probably vary from application to application.

If predeclaration is in effect, Methods 10 and 12 (centralized 2PL for rw , primary copy 2PL or centralized 2PL for ww) should also be considered. There are two cases to analyze.

1. If condition A of Section 6.2.4 also holds -- i.e., if the average transaction reads data from more than two DMs at which it does not write -- then Method 12 dominates the other choices insofar as communication and computation are concerned; see figure 6.20. For this case, Method 12 (centralized 2PL for all synchronization) is optimal.
2. If condition A does not hold, the choice of methods again involves a tradeoff between the extra locks required by Method 2 vs. the extra messages required by Methods 6, 10, and 12.

The choice of deadlock resolution technique is not critical in optimistic applications because few conflicts, and even fewer deadlocks are expected. The designer is free to select whatever technique is easiest to implement in his system.

Mixed methods are also feasible for optimistic applications. However these methods are dominated by 2PL methods insofar as communication and computation overhead are concerned; see figure 6.20. Therefore, mixed methods should probably not be considered in optimist situations.

The analysis of this section may be summarized as follows.

(i) T/O Method 8 (basic T/O for rw, with TWR for ww) is the best T/O method for optimistic applications.

(ii) All dominant 2PL methods are reasonable choices under one or more application scenarios.

(iii) The choice among 2PL methods and between these methods and T/O Method 2 involves a tradeoff between communication and computation overhead. The best method will probably be different for different applications, although all of these methods will probably perform well.

(iv) Mixed methods are not appropriate for optimistic applications.

6.7.3 Handling In-Between Applications

Optimism and pessimism are end-points of a spectrum. Undoubtedly, most applications will lie somewhere in the middle of this spectrum. For these applications all cost factors have a potentially important impact on concurrency control performance. In the absence of quantitative performance data, there is little more to be said.

6.8 Review of Past Work

The state-of-the-art in performance analysis of distributed concurrency control algorithms is represented by two recent PhD theses [G-M, Ries]. Each author compared the performance of a few algorithms under limited operating conditions.

Garcia-Molina's Analysis

Garcia-Molina compared several variants of centralized 2PL (i.e., 2PL Method 12) to two distributed algorithms, namely the Majority Consensus Algorithm of [Thomas 1,2] and The Ring Algorithm of [Ellis]. The Thomas and Ellis algorithms do not correspond to any of our methods. We chose not to include these algorithms in our framework because, as we explain in the Appendix, these algorithms are inherently inefficient. The analysis carried out by Garcia-Molina in [G-M 2] supports this conclusion; the author concluded centralized 2PL out-performs the [Ellis] and [Thomas] algorithms under all tested conditions.

The analysis in [G-M 2] depends upon several restrictive assumptions.

(i) Predeclaration of readsets and writesets is assumed.

(ii) It is assumed that the readset of a transaction always subsumes its writeset; i.e., a transaction cannot write into a logical data item X unless it also reads X. This assumption qualitatively reduces the complexity of the concurrency control problem [BSW].

(iii) A fully redundant database is assumed -- this means that every DM is assumed to contain a copy of every logical data item in the database. A number of the refinements analyzed by Garcia-Molina cannot be implemented without this assumption.

(iv) Very small readsets were assumed -- much of the analysis considered readsets of 5 data items.

(v) Last, but not least, the combinations of operating parameters considered in the analysis were such that run-time conflicts almost never occurred. In other words, the analysis was limited to optimistic situations.

Ries's Analysis

The analysis in [Ries 1] compares two variations of centralized 2PL (i.e., 2PL Method 12) to two variations of distributed 2PL (i.e., 2PL Method 1). For all variations, predeclaration is assumed. For the centralized 2PL algorithms, pre-ordering of resources is also assumed; this is not assumed for the distributed 2PL algorithms. The two centralized algorithms differ in the following respect: in variation 1, transactions are blocked if they request unavailable locks, while in

variation 2 they are restarted. The two distributed algorithms differ in their deadlock resolution scheme: variation 1 uses the Wait-Die deadlock prevention technique (see Section 4.17) while variation 2 uses centralized deadlock detection.

The operating conditions considered by [Ries 1] were such that run-time conflicts rarely occurred. Indeed, no deadlocks ever arose in the study. Thus, Ries's analysis, like Garcia-Molina's, limited its attention to optimistic situations. In addition, transactions tended to read and write data at the same sites. This means that condition A of Section 6.2.4 tended not to hold under the operating conditions that were considered.

The main result of [Ries 1] is that all four concurrency control algorithms exhibit similar behavior. No algorithm is clearly better than the others, and the differences between algorithms are slight under almost all operating conditions. This null result lends support to our analysis of optimistic situation in Section 6.7.2; our qualitative analysis indicated that in optimistic situations where condition A of Section 6.2.2 does not hold, centralized 2PL and distributed 2PL* are both feasible and neither dominates the other. It seems probable to us that any variations in performance observed by Ries were caused more by differences in deadlock resolution technique than by differences in principal method.

The work of Garcia-Moline and Ries are important first steps in concurrency control performance analysis. This work emphasizes the difficultly (perhaps futility) of attempting such analysis without first structuring the space of potential solutions to the distributed concurrency control problem.

---

*The distributed 2PL method considered in Section 6.7.2 is Method 2 whereas Ries considered Method 1. The performance of these methods only differs with respect to restart behavior; see figure 6.12. Restart behavior is not important in optimistic situations.

7. State-Of-The-Art and Directions for Further Work

The long range goal of research on distributed database concurrency control is to develop a methodology for designing good concurrency control methods for a given system environment and a given class of applications. As a first step toward this goal, we have studied the state-of-the-art of distributed database concurrency control and produced an integrated overview of that subject. In this section, we summarize the findings of our study, and outline recommendations for future research.

7.1 Summary

We set the stage for our study of distributed concurrency control in Section 2. That section presented a model of transaction processing in a distributed DBMS emphasizing the essential inter-site interactions needed to process users' transactions. This model provided a common framework for describing and analyzing concurrency control methods, a framework that has been lacking in the literature.

In Section 3 we reviewed the mathematical theory of concurrency control. We formalized the principal correctness criterion for

a concurrency control method -- namely serializability. And we showed how the overall problem of attaining serializability can be decomposed into two sub-problems -- read-write and write-write synchronization. This decomposition is the cornerstone of our paradigm for the design and analysis of concurrency control methods.

Section 4 exploited this paradigm to examine fundamental read-write and write-write synchronization techniques outside the context of any specific concurrency control method. We considered virtually all known synchronization techniques and showed that these techniques can be understood as variations of two basic techniques -- two-phase locking (2PL) and timestamp ordering (T/O). This consolidation of the state-of-the-art was possible in large part because of the paradigm for concurrency control introduced in Section 3 and the common transaction processing model introduced in Section 2.

In Section 5 we studied the space of concurrency control methods that can be constructed from the techniques of Section 4. We found this space to be enormous: there are thousands of ways of combining the synchronization techniques of Section 4 into complete concurrency control methods. However, we identified 48 of these methods as principal methods. Most of these principal methods have not been described in the literature previously, and several of these new methods have interesting performance characteristics.

Section 6 was devoted to performance analysis of principal methods. We described the four main performance measures for concurrency control methods -- communication overhead, local processing overhead, transaction restarts, and transaction blocking. We analyzed the major synchronization techniques and the principal concurrency control methods relative to each performance measure in qualitative terms. We showed that no method had optimal performance under all four measures, which means that no method can be expected to perform optimally for all system environments and applications. However, we identified 11 methods as dominant methods -- for any given system and application we believe that one of these 11 dominant methods will be optimal. In addition, we suggested a design scenario for choosing among dominant methods for certain kinds of stereotypical applications.

Finally, in the Appendix we discussed three methods that have appeared in the literature but do not fit into the framework of Sections 4 and 5. While these methods are intellectually impressive (and, in some cases, famous), none is of practical significance in a distributed database environment.

Briefly, then, the state-of-the-art in distributed database concurrency control is as follows.

1. a large number of correct methods for distributed database concurrency control are known;
2. many important characteristics of system environments and applications have been isolated;
3. the relative performance of concurrency control methods has been qualitatively analyzed for some combinations of system and application characteristics.

Lacking empirical test data, our knowledge of concurrency control performance is necessarily qualitative. Even though we cannot predict how much better one concurrency control method is than another, this qualitative knowledge is nonetheless very important. The strong demand for distributed database systems makes their development inevitable, however little we may know about the performance of distributed concurrency control. Our qualitative understanding can help considerably in the design of such systems, by providing a structured range of alternatives from which to select concurrency control methods and some intuition as to which methods are best. Thus, the first tools for thought about the design of distributed concurrency control methods are in place.

7.2 Recommendations

We recommend that the next stage of reserch expand both our qualitative and quantitative knowledge of distributed database concurrency control, with an eye toward assisting the distributed DBMS designer in the selection of a concurrency control method. Specifically, we recommend that aspects of distributed data management that are related to and affect the performance of concurrency control be investigated, especially distributed reliability and distributed database design. In addition, given the large number of alternative concurrency control methods, it is appropriate that future research be directed toward pruning the range of choices to simplify the selection of best methods. To begin the pruning process, we recommend that basic performance data on concurrency control methods for particular applications and system environments be obtained. We discuss each of these problem areas below.

7.2.1 Reliability

The performance of concurrency control methods is influenced by reliability considerations for two reasons. First, the communication requirements of a reliability protocol can affect the communication requirement of a concurrency contol method. Two-phase commit is an example of this effect. Second, different concurrency control methods may be susceptible to different kinds of failures and may need different types of algorithms to make them reliable. When reliability is an important design goal, we must include the cost of making a concurrency control method reliable when evaluating the method's performance. Overall, reliability issues are intimately related to, but less well understood than, concurrency control methods and should be investigated more thoroughly.

7.2.2 Distributed Database Design

The performance of distributed concurrency control methods is likely to be sensitive to the design of a distributed database. The number of sites over which the database is spread and the number of redundant copies that are maintained will both

influence, and be influenced by, the selection of a concurrency control method. Research on distributed database design has concentrated on query processing issues, almost totally ignoring concurrency control factors [RG 2]. We therefore recommend that the relationship between distributed database design and distributed database concurrency control be explored.

7.2.3 Basic Performance Data

To obtain basic performance data, application and system characteristics must be expressed in quantifiable parameters. It must be possible to estimate these parameters given a rudimentary description of a system and an application, since that is the kind of description typically available when a concurrency control method must be selected.

Given this parameterization, performance data should be obtained to validate and expand the assumptions and conclusions presented in Section 6. To simplify the analysis, synchronization techniques should be examined in isolation, outside of a system environment. The goal of this analysis should be two-fold: first, to determine the relative performance of synchronization techniques for a wide range of system and application behavior; and second, to determine which of the four major performance factors -- local processing, blocking, restarts, and

communication -- are most critical in different system and applications environments. Both types of data will further structure the problem of selecting best concurrency control methods.

We also recommend that the importance of concurrency control performance in a distributed DBMS be studied, to determine when concurrency control is a bottleneck. We expect that for some systems and applications, concurrency control induces an insignificant run-time cost. In such cases, the best concurrency control method may be the one that is easiest to implement.

7.2.4 Final Remarks

The research we have recommended will yield fundamental data regarding the performance of concurrency control methods, and will explore the effects of other aspects of distributed database management on concurrency control performance. Early results on these problems may well suggest the direction and approach to subsequent research. Until our understanding of these problems is deepened, we believe it inappropriate to perform detail analyses of specific methods.

The basic investigations we have proposed will, in the short run, assist in the design of concurrency control methods for practical systems and, in the longer term, help develop a comprehensive methodology for concurrency control design and analysis.

Acknowledgements

This study reflects accumulated discussions with many colleagues over several years. In this regard, we would especially like to thank Marco Casanova, Dave Shipman, Jim Rothnie, and Dushan Badal. We also greatly appreciate the assistance of John Smith who, as project manager, served both as our chief sounding board and critical reviewer for this document.

We also gratefully acknowledge the tireless dedication of Debbie Huebner who typed, editted, and text-processed this manuscript.

A. Other Concurrency Control Methods

In this appendix we describe three concurrency control methods that do not fit into the framework of Sections 4 and 5. These are the certifier methods of [Badal 2] and [Cassanova], the majority consensus algorithm of [Thomas 1,2], and the ring algorithm of [Ellis]. We shall argue that none of these methods are practical in distributed database environments. The certifier methods have been invented fairly recently and look promising for centralized database system. However, as we will describe, severe technical problems must be overcome before these methods can be extended correctly to distributed systems. The [Thomas] and [Ellis] algorithms, by contrast, are among the earliest algorithms proposed for distributed database concurrency control. These algorithms introduced several important techniques into the field, but as we will see, they have been surpassed by recent developments.

A.1 Certifiers

A.1.1 The Certification Approach

In the certification approach, dm-reads and pre-commits are processed by DMs first-come-first-served, with no synchronization whatsoever. DMs do maintain summary information about rw and ww conflicts, which they update every time an operation is processed. However, dm-read and pre-commits are never blocked or rejected based on the discovery of such a conflict.

Synchronization occurs at the time a transaction attempts to terminate. When a transaction, T, issues its END operation, the DBMS decides whether or not to certify and thereby commit T.

To understand how this decision is made, we must distinguish between "total" and "committed" executions. A total execution of transactions includes the execution of all operations processed by the system up to a particular moment. The committed execution is the portion of the total execution that only includes dm-reads and dm-writes processed on behalf of

committed transactions. That is, the committed execution is the total execution that would result from aborting all active transactions (and not restarting them).

When T issues its END operation, the system tests whether the committed execution augmented by the execution of T is serializable. That is, it tests whether after committing T the resulting committed execution would still be serializable. If the answer is "yes", T is committed; otherwise T is restarted.

There are two properties of certification that distinguish it from other approaches. First, synchronization is accomplished entirely by restarts, never by blocking. And second, the decision to restart or not is made after the transaction has finished executing. None of the concurrency control methods we discussed in Sections 4 and 5 satisfy both these properties.

The rationale for using certification is based on an optimistic attitude regarding run-time conflicts. The argument is that if very few run-time conflicts are expected, we might as well assume that most executions are serializable. By processing dm-reads and pre-commits without synchronization, the concurrency control method never delays a transaction while it is being processed. A (hopefully fast) certification test when the transaction terminates is all that is required. Assuming optimistic transaction behavior, we expect that the test will usually result in committing the transaction, so there are very

few restarts. Therefore, certification simultaneously avoids blocking and restarts in optimistic situations.

A certification concurrency control method must include a summarization algorithm for storing information about dm-reads and pre-commits when they are processed and a certification algorithm for using that information to certify transactions when they terminate. The main problem in the summarization algorithm is avoiding the need to store information about transactions that have already been certified. The main problem in the certification algorithm is obtaining a consistent copy of the summary information. To obtain a consistent copy of the summary information, it is often necessary for the certification algorithm perform some synchronization of its own; the cost of that synchronization must be included in the cost of the entire method.

A.1.2 Certification Using the -> Relation

One certification method is to construct the -> relation as dm-reads and pre-commits are processed. To certify a transaction, the system checks that the -> relation has no cycles [Badal 2] [Cassanova].

To construct the -> relation, each site remembers the most recent transaction that read or wrote each data item. Suppose transactions T_i and T_j were the last transactions to (respectively) read and write data item x. If transaction T_k now issues a dm-read(x), T_j -> T_k is added to the summary information for the site and T_k replaces T_i as the last transaction to have read X. Similarly, if T_k issues a pre-commit(x), T_i -> T_k and T_j -> T_k are added to the summary and T_k replaces T_j as the last transaction to have written x. Thus, pieces of the -> relation are distributed among the sites, reflecting run-time conflicts at each site.

To certify a transaction, the system must check that the transaction does not lie on a cycle in -> (see Theorem 1', in Section 3). This is sufficient to guarantee serializability.

There are two problems with this approach. First, it is in general not correct to delete a certified transaction from -> , even if all of its updates have been committed. For example, if T_i -> T_j and T_i is active but T_j is committed, it is still possible for T_j -> T_i to develop; deleting T_j would then cause the cycle T_i -> T_j -> T_i to go unnoticed when T_i is being certified. However, it is obviously not feasible to allow -> to grow indefinitely. This problem is solved in [Casanova] by a method of encoding information about committed transactions in space proportional to the number of active transactions.

A second problem is that all sites must be checked to certify any transaction. Even sites at which the transaction never accessed data must participate in the cycle checking of ->. For example, suppose we want to certify transaction T. T might be involved in a cycle T -> T_1 T_2 ->...-> T_{n-1} -> T_n -> T_1, where each conflict T_k -> T_{k+1} occurred at a different site. Possibly, T only accessed data at one site, yet the -> relation must be examined at n sites to certify T. This problem is currently unsolved, as far as we know. That is, any correct certifier based on this approach of checking cycles in -> must access the -> relation at all sites to certify each and every transaction. Until this problem is solved, we judge the certification approach to be impractical in a distributed environment.

A.2 Thomas' Majority Consensus Algorithm

A.2.1 The Algorithm

One of the first published algorithms for distributed concurrency control is a certification method described in [Thomas 1,2]. Thomas introduced several important synchronization techniques in that algorithm including the Thomas Write Rule (see Section 4.2.3), majority voting (see Section 4.1.1), and certification (see Appendix A.1). These techniques are valuable when considered in isolation. However, we will argue that the overall Thomas algorithm is not suitable for distributed databases. We begin by describing the algorithm and then comment on its application to distributed databases.

Thomas' algorithm assumes a fully redundant database. That is, each logical data item is stored at every site. Also, each copy carries the timestamp of the last transaction that wrote into it.

Transactions execute in two-phases. In the first phase, each transaction executes locally at one site called the transaction's home site. Since the database is fully redundant, any site can serve as the home site for any transaction. The transaction is assigned a unique timestamp when it begins executing. During execution, it keeps a record of the timestamp

of each data item it reads. When the transaction executes a write on a data item, the system processes the write by recording the new value in an update list. Importantly, each transaction is required to read a copy of a data item before it writes into that data item. When the transaction terminates, the system augments the update list with the list of data items read by the transaction and the timestamps of those data items at the time they were read. In addition, the timestamp of the transaction itself is added to the update list. This completes the first phase of execution.

In the second phase, the update list is sent to every site. Each site (including the site that produced the update list) votes on the update list. Intuitively speaking, to vote on an update list, a site tries to certify the transaction that produced the list. The site votes yes iff it can certify the transaction. After a site votes yes on an update list, the update list is said to be pending at that site. To cast the vote, the site sends a message to the transaction's home site. When the home site receives a majority of yes or no votes, it informs all sites of the outcome of the vote. If a majority voted yes, then all sites are required to commit the update, which are then installed using the Thomas Write Rule. If a majority voted no, all sites are told to discard the update, and the transaction is restarted.

The rule that determines when a site may vote "yes" on a transaction is pivotal to the correctness of the algorithm. To vote on an update list, U, a site compares the timestamp of each data item in the readset of U to the timestamp of that same data item in the site's local database. If any data item has a different timestamp in the database than in U, the site votes no. If all data items satisfy the timestamp comparison, the site compares the readset and writeset of U to the readset and writeset of each pending update list at that site. If there is no rw conflict between U and any of the pending update lists, the site votes yes. If there is an rw conflict between U and one of those pending requests, the site votes pass (which amounts to abstaining) if the timestamp of U is larger than the timestamp of the pending update list with which it conflicts. If there is an rw conflict and U's timestamp is smaller than the timestamp of the pending update list with which it conflicts, then it sets U aside on a wait queue and tries again to vote on U as soon as the request with which U conflicts has either been committed or aborted at that site.

The voting rule is essentially a certification procedure. By making the timestamp comparison, a site is checking that the readset was not written into since the transaction read that readset. If the comparisons are satisfied, the situation is as if the transaction had locked its readset at that site and held the locks until the time it voted. So, the voting rule is

guaranteeing rw synchronization with a certification rule approximating rw locking. (This fact is proved precisely in [BSW].)

The second part of the voting rule, in which U is checked for rw conflicts against pending update lists, guarantees that conflicting requests are not certified concurrently. An example illustrates the problem. Suppose transaction T1 reads X and Y, and writes Y, while transaction T2 reads X and Y, and writes X. Suppose T1 and T2 execute at sites A and B respectively and X and Y have timestamps of 0 at both sites. Assume that T1 and T2 execute concurrently and produce update lists ready for voting at about the same time. Notice that either T1 or T2 must be restarted, since neither read the other's output; if they were both committed the result would be non-serializable. However, both T1's and T2's update lists will (concurrently) satisfy the timestamp comparison at both A and B. What stops them from both obtaining unanimous yes votes is the second part of the voting rule. After a site votes on one of the transactions, it is prevented from voting on the other transaction until the first is no longer pending. The point of the example is that conflicting transactions cannot be concurrently certified without violating serializability.

With the second part of the voting rule, the algorithm behaves as if the certification step were atomically executed at a

primary site. If certification were centralized at a primary site, the certification step at the primary site would serve the same role as the majority decision in the voting case.

(We note, that this problem of concurrent certification exists in the algorithms of A.1.2, too. This is yet another technical difficulty with the certification approach in a distributed environment.)

A.2.2 Correctness

No simple proof of the serializability of Thomas' algorithm has ever been demonstrated, although Thomas provided a detailed "plausibility" argument in his paper [Thomas 2]. The first part of the voting rule can correctly be used in a centralized concurrency control method since it implies 2PL [BSW]. A centralized method based on this approach was proposed in [KR]. The second part of the voting rule guarantees that for every pair of conflicting transactions that received a majority of yes votes, all sites that voted yes on both transactions voted on the two transactions in the same order. This makes the certification step behave just as it would if it were centralized, thereby avoiding the problem exemplified in the previous paragraph.

A.2.3 Partially Redundant Databases

For the majority consensus algorithm to be useful in a distributed database environment, the algorithm must be generalized to operate correctly when the database is only partially redundant. There is reason to doubt that such a generalization can be accomplished without either serious degradation of performance or a complete change in the set of techniques that are used.

First, the majority consensus decision rule apparently must be dropped. The voting algorithm fundamentally depends upon the fact that all sites are performing exactly the same certification test. In a partially redundant database, each site would only be comparing the timestamps of those data items stored at that site. Thus, the significance of the majority vote would vanish.

If majority voting cannot be used to synchronize concurrent certification tests, apparently some kind of mutual exclusion mechanism must be used instead. The purpose of the mutual exclusion mechanism would be to prevent the concurrent, and therefore potentially incorrect, certification of two conflicting transactions. Such a mutual exclusion mechanism

would amount to locking. The use of locks for synchronizing the certification step violates the spirit of Thomas' algorithm, since a main goal of the algorithm was to avoid the need for locking. However, it is worth examining such a locking mechanism to see how certification can be correctly accomplished in a partially redundant database.

To process a transaction T, a site produces an update list exactly as in the fully redundant case. However, since the database is partially redundant, it may be necessary to read portions of T's readset from other sites. After T terminates, its update list is sent to every site that contains part of T's readset or writeset. To certify an update list, a site first sets local locks on the readset and writeset. Then it compares the update list's timestamps to the database's timestamps, as in the fully redundant case. If they are identical, it "votes" yes otherwise, it "votes" no. A unanimous vote of yes is needed to commit the updates. Local locks cannot be released until the voting decision is completed.

While this version of Thomas' algorithm for partially redundant data works correctly, its performance is inferior to standard 2PL. This algorithm requires that the same locks be set as in 2PL and the same deadlocks can arise. Yet the probability of restart is higher than in 2PL, because even after all locks are obtained the certification step can still vote no (which 2PL never does).

One can improve this algorithm by designating a primary copy of each data item and only performing the timestamp comparison against the primary copy. This is analogous to primary copy 2PL. However, for the same reasons as above, we would expect primary copy 2PL to outperform this version of Thomas' algorithm too.

We therefore must leave open the problem of producing an efficient version of Thomas' algorithm for a partially redundant database.

A.2.4 Performance

Even in the fully redundant case, the performance of the majority consensus algorithm is not very good. First, repeating the certification and conflict detection at each site is more than what is needed to obtain serializability. A centralized certifier would work just as well and would only require that certification be performed at one site. Second, the algorithm is quite prone to restarts when there are run-time conflicts, since restarts are the only tactic available for synchronizing transactions. (Other certification techniques have this problem as well; see Appendix A.1.) Hence, the algorithm will only perform well under the most optimistic circumstances. Finally, even in optimistic situations, the analysis in [G-M 2] indicates

that centralized 2PL outperforms the majority consensus algorithm.

A.2.5 Reliability

Despite the performance problems of the majority consensus algorithm, one can try to justify the algorithms on reliability grounds. As long as a majority of sites are correctly running, the algorithm runs smoothly. This means that there is no cost in handling a site failure, insofar as the voting procedure is concerned.

However, based on current knowledge, this justification is not compelling for several reasons. First, although there is no cost when a site fails, substantial effort may be required when a site recovers. A centralized algorithm using back-up sites, as in [AD], lacks the symmetry of Thomas' algorithm, but may well be more efficient due to the simplicity of site recovery. In addition, the majority consensus algorithm does not consider the problem of atomic commitment and it is unclear how one would integrate two-phase commit into the algorithm.

Overall, the reliability threats that are handled by the majority consensus algorithm have not been explicitly listed, and alternative solutions have not been analyzed. While voting

is certainly an interesting technique for obtaining a measure of reliability, the circumstances under which it is cost-effective are unknown.

A.3 Ellis' Ring Algorithm

Another early solution to the problem of distributed database concurrency control is the ring algorithm [Ellis]. Ellis was principally interested in a proof technique, called L systems, for proving the correctness of concurrent algorithms. He developed his concurrency control method primarily as an example to illustrate L system proofs, and never made claims as to its performance. Because the algorithm was only intended to illustrate mathematical techniques, Ellis imposed a number of restrictions on the algorithm for mathematical convenience, which happen to make it infeasible in practice. Nonetheless the algorithm has received considerable attention in the literature, and in the interest of completeness, we briefly discuss it.

Ellis' algorithm solves the distributed concurrency control problem with the following restrictions:

1. The database must be fully redundant.
2. The communication medium must be configured as a ring, so each site can only communicate with its successor on the ring.
3. Each site-to-site communication link is pipelined.

4. Each site can supervise no more than one active update transaction at a time.
5. To update any copy of the database, a transaction must first obtain a lock on the entire database at all sites.

The effect of restriction 5 is to force all transactions to execute serially; no concurrent processing is ever possible. For this reason, the algorithm is fundamentally impractical.

To execute, an update transaction migrates arond the ring, (essentially) obtaining a lock on the entire database at each site. However, the lock conflict rules are nonstandard. A lock request from a transaction that originated at site A conflicts at site C with a lock held by a transaction that originated from site B if B=C and either A=B or the priority of site A is less than the priority of site B. The daisy chain comunication induced by the ring combined with this locking rule produces a deadlock-free algorithm that does not require deadlock detection and never induces restarts. A detailed description of the algorithm appears in [G-M].

There are several problems with this algorithm in a distributed database environment. First, as mentioned above, it forces transactions to execute serially. Second, it only applies to a fully redundant database. And third, the daisy chain communication requires that each transaction obtain its lock at

one site at a time, which causes communication delay to be (at least) linearly proportional to the number of sites in the system.

A modified version of Ellis' algorithm that mitigates the first problem is proposed in [G-M 2]. Even with this improvement, Garcia-Molina's performance analysis indicates that the ring algorithm has inferior performance to centralized 2PL. And, of course, the modified algorithm still suffers from the last two problems.

References

[AD]
Alsberg, P.A. and Day, J.D. "A Principle for Resilient Sharing of Distributed Resources", Proc. 2nd Int. Conference on Software Engineering, October 1976.

[ABDG]
Alsberg, P.A. Belford, G.G. Day, J.D. and Grapa, E. "Multi-Copy Resiliency Techniques," Center for Advanced Computation, AC Document No. 202, University of Illinois at Urbana-Champaign, Urbana Illinois, May 1976.

[AHU]
Aho, A.V., Hopcroft, E., Ullman, J.D. "The Design and Analysis of Computer Algorithms." Addison-Wesley Publishing Co. (1975).

[Badal 1]
Badal, D.Z. University of California at Los Angeles, Los Angeles, California. "On Efficient Monitoring of Data Base Aggertions in Distributed Data Base Systems." Proc. 4th Berkeley Conference on Distributed Data Management & Computer Networks. August 1979.

[Badal 2]
Badal, D. Z. "Correctness of Concurrency Control and Implications in Distributed Databases", Proc. COMPSAC 79 Conf., Chicago, Nov. 1979.

[Badal 3]
Badal, D. Z. "On the Degree of Concurrency Provided by Concurrency Control Mechanisms for Distributed Databases", Proc. the Intl. Symposium on Distributed Databases, Versailles, France, Mach 1980 (to appear).

[Badal 4]
Badal, D. Z. "Integrity, Consistency and Concurrency in Distributed Database Systems", (invited paper), Infotech State of the Art Report on Distributed Databases, Infotech Ltd., Maidenhead, UK, July 1979.

[BP]

Badal, D.Z.; and Popek, G.J. "A Proposal for Distributed Concurrency Control for Partially Redundant Distributed Data Base System," Proc. 3rd Berkeley Workshop on Distributed Data Management and Computer Networks, 1978, pp. 273-288.

[BKZ]

Bamino, J.S., C. Kaiser, and H. Zimmerman, "Synchronization for Distributed Database Systems Using a Single Broadcast Channel", Proc. First International Conf. on Distributed Computing Systems, IEEE, N.Y., Oct. 1979, pp. 330-338.

[BSS]

Belford, G. C., Schwartz, P. M. and Sluizer, S. The Effect of Back-up Strategy on Database Availability, CAC Document No. 181, CCTCWAD Document No. 5515, Center for Advanced Computation, University of Illinois at Urbana-Champaign, February 1976.

[BCG]

Bernstein, P.A., M.A. Casanova, and N. Goodman, "Comments on Process Synchronization in Database Systems", ACM Trans. on Database Sys., Vol. 4, No. 4, Dec. 1979.

[BRGP]

Bernstein, P.A., Rothnie, J.B., Goodman, N. and Papadimitriou, C.H. "The Concurrency Control Mechanism of SDD-1: A System for Distributed Databases (The Fully Redundant Case)", IEEE Trans. on Software Engineering, Vol. SE-4, No. 3 (May 1978).

[BSJ]

Bernstein, P.A. and Shipman D.W. "A Formal Model of Concurrency Control Mechanisms for Database Systems," Proc. 1978 Berkeley Workshop on Distributed Databases and Computer Networks.

[BS 2]

Bernstein P. and Shipman D. "The Correctness of Concurrency Mechanisms in a System for Distributed Databases (SDD-1)" ACM Trans. on Database Systems, Vol. 5, No. 1, March 1980.

[BSR]

Bernstein P., Shipman D., and Rothnie J. "Concurrency Control in a System for Distributed Databases (SDD-1)" ACM Trans. on Database Systems, Vol. 5, No. 1, March 1980.

[BSW]
Bernstein, P. A., Shipman D. W. and Wong, W. S. "Formal Aspects of Serializability in Database Concurrency Control", IEEE Trans. on Software Engineering, Vol. SE-5, No. 3, May 1979.

[BK]
Breitwieser, H. and Kersten, U. Tech. Hoch. Karlsruhe, W. Germany "Transaction and Catalog Management of the Distributed File Management System DISCO," Proc. VLDB 79, Rio de Janeiro

[Casanova]
Casavona, M.A., "The Concurrency Control Problem for Database Systems", Ph.D. Thesis, Harvard University, Technical Report TR-17-79, Center for Reserch in Computing Technology, 1979.

[CBT]
Chamberlin, D. D., Boyce, R. F. and Traiger, I. L. "A Deadlock-Free Scheme for Resource Allocation in a Database Environment," Info. Proc. 74, North¬Holland, Amsterdam, 1974.

[CY]
Cohen, D. and Yemini, Y. USC-ISI, Marina del Rey, California "Protocols for Dating Coordination," Proc. 4th Berkeley Conference on Distributed Data Management & Computer Networks, August 1979.

[Date]
Date, C.J. "An Introduction to Database Systems." Addison-Wesley Publishing Co. (1977).

[DF]
Deppe, M. E. and Fry, J. P. "Distributed Databases: A Summary of Research," Computer Networks, Vol. 1, No. 2, North¬Holland, Amsterdam, Sept. 1976.

[Ellis]
Ellis, C.A. "A Robust Algorithm for Updating Duplicate Databases," Proc. 2nd Berkeley Workshop on Distributed Databases and Computer Networks, May 1977.

[ESW]
Epstein, R., M. Stonebraker, and E. Wong, "Distributed Query Processing is a Relational Database System" Proc. 1978 ACM-SIGMOD Conference, ACM, N.Y., May 1978.

[EGLT]
Eswaran, K.P., Gray, J.N., Lorie, R.A. and Traiger, I.L. "The Notions of Consistency and Predicate Locks in a Database System", Communications of the ACM, Vol. 19, No. 11, November 1976.

[Everest]
Everest, G. C. "Concurrent Update Control and Database Integrity," in Database Management, Klimbie, J. W. and Koffeman, K. L. (eds.), North-Holland, Amsterdam, 1974.

[G-M 1]
Garcia-Molina, H. "Performance Comparisons of Two Update Algorithms for Distributed Databases," Proc. 3rd Berkeley Workshop on Distributed Databases and Computer Networks, August 1978.

[G-M 2]
Garcia-Molina, H. "Performance of Update Algorithms for Replicated Data in a Distributed Database", Ph.D. Dissertation, Computer Science Department, Stanford University, June 1979.

[G-M 3]
Garcia-Molina, H. Stanford University, Stanford, California "A Concurrency Control Mechanism for Distributed Data Bases Which Uses Centralized Locking Controllers," Proc. 4th Berkeley Conference on Distributed Data Management & Computer Networks, August 1979.

[G-M 4]
Garcia-Molina, H., "Centralized Control Update Algorithms for Fully Redundant Distributed Databases", Proc. First International Conf. on Distributed Computing Systems, IEEE N.Y., Oct. 1979, pp. 699-705.

[GL]
Gardarin, G. and Lebaux, P. "Scheduling Algorithms for Avoiding Inconsistency in Large Databases." Proc. 1977 Int. Conf. on Very Large Data Bases,IEEE, N.Y., 501-516

[GS]
Gelembe, E. and Sevcik, K. "Analysis of Update Synchronization for Multiple Copy Databases," Proc. 3rd Berkeley Workshop on Distributed Databases and Computer Networks, August 1978.

[GBWRR]

Goodman, N., P.A. Bernstein, E. Wong, C.L. Reeve, and J.B. Rothnie, "Query Processing in SDD-1: A system for Distributed Databases", Technical Report CCA-79-06, Computer Corporation of America, Cambridge, MA, Oct. 1979 (submitted for publication).

[GBW]
Gouda, M., Boyd, D. and Wood, W. Honeywell, Bloomington, Minnesota "Global and Local Models for the Specification and Verification of Distributed Systems," Proc. 4th Berkeley Conference on Distributed Data Management & Computer Networks, August 1979.

[Gouda]
Gouda, M.G. "A Hierarchical Controller for Concurrent Accessing of Distributed Databases", Proc. 4th Workshop on Computer Architecture for Non-Numeric Processing, August 1978.

[GLPT]
Gray, J. N., Lorie, R. A., Putzulo G. R. and Traiger, I. L. "Granularity of Locks and Degrees of Consistency in a Shared Database," IBM Research Report RJ1654, September 1975.

[Gray]
Gray, J. N. Notes on Database Operating Systems, unpublished lecture notes. IBM San Jose Research Laboratory, San Jose, Calif., 1977.

[Hewitt]
Hewitt, C.E., "Protection and Synchronization in Actor Systems", Working Paper 83, MIT Artificial Intelligence Lab, Nov. 1974.

[Hoare]
Hoare, C.A.R., "Monitors: An Operating System Structure Concept", CAMC 17, 10 (Oct 1974), 549-557.

[HS 1]
Hammer, M. M. and Shipman, D. W. "An Overview of Reliability Mechanisms for a Distributed Data Base System," Proc. 1977 COMPCON, IEEE, N.Y.

[HS 2]
Hammer, M.M., and Shipman, D.W., "Reliability Mechanisms for SDD-1:
A System for Distributed Datases", Technical Report CCA-79-05, Computer Corporation of America, Cambridge, MA, 1979 (submitted for publication).

[HSW]
Helb, G., M. Stonebraker, E. Wong, "INGRES-A Relational Database System", Proc. 1975 National Computer Conf., AFIPS Press, Montrale, N.J.

[HY]
Henver, A.R., and S.B. Yao, "Query Processing in Distributed Databases", IEEE Trans. on Soft. Eng., Vol SE-5, No. 3, May 1979.

[HV]
Herman, D. and J.P. Verjus, "An Algorithm for Maintaining the Consistencyof Multiple Copies", Proc. First International Conf. on Distributed Computing Systems, IEEE, N.Y., pp. 625-631.

[JT]
Johnson, P.R. and Thomas, R.H. The Maintenance of Duplicate Databases", Network Working Group RFC #677 NIC #31507, January 27, 1975.

[KNTH]
Kaneko, A., Y.Nishihara, K. Tsuruoka, and M. Hattori, "Logical Clock Synchronization Method for Duplicated Database Control", Proc. First International Conf. on Distributed Computing Systems, IEEE, N.Y., Oct. 1979, pp. 601-611.

[KMIT]
Kawazu, S., Minami, S., Itoh, K. and Teranaka, K. "Two-Phase Deadlock Detection Algorithm in Distributed Databases," Proc. 1979 International Conference on Very Large Data Bases, IEEE, N.Y.

[KC]
King, P. F., and Collmeyer, A. J. "Database Sharing--An Efficient Mechanism for Supporting Concurrent Processes," Proc. 1974 NCC, AFIPS Press, Montvale, New Jersey, 1974.

[KP]
Kung, H.T. and Papadimitriou, C.H. "An Optimality Theory of Concurrency Control for Databases." Proc. 179 ACM-SIGMOD Int. Conf. on Management of Data (June 1979).

[KR]
Kung, H.T. and Robinson, J.T. "On Optimistic Methods for Concurrency Control." Proc. 1979 Int. Conf. on Very Large Data Bases (Oct. 1979).

[Lamport 1]
Lamport, L. Time, Clocks, and the Ordering of Events in a Distributed System, Massachusetts Computer Associates, CA-7603-2911, Wakefield, Mass., March 1976.

[Lamport 2]
Lamport, L. Towards a Theory of Correctness of Multi-User Database Systems, Massachusetts Computer Associates, CA-7610-0712, October 1976.

[LS]
Lampson, B. and Sturgis, H. Crash Recovery in a Distributed Data Storage System, Tech. Report, Computer Science Laboratory, Xerox Palo Alto Research Center, Palo Alto, Calif. 1976.

[Lelann]
LeLann, G. "Algorithms for Distributed Data-Sharing Systems Which Use Tickets," Proc. 3rd Berkeley Workshop on Distributed Databases and Computer Networks, August 1978.

[Lin]
Lin, W. K. Sperry Research Center, Sudbury, Massachusetts "Concurrency Control in a Multiple Copy Distributed Data Base System," Proc. 4th Berkeley Conference on Distributed Data Management & Computer Networks, August 1979.

[MM]
Menasce, D.A. and Muntz, R.R. "Locking and Deadlock Detection in Distributed Databases," IEEE Transactions on Software Engineering, Vol SE-5, No. 3, May 1979, pp. 195-202.

[MPM]
Menasce, D.A., G.J. Popek and R.R. Muntz "A Locking Protocol for Resource Coordination in Distributed Databases", Proc. 1978 ACM-SIGMOD Conf. on Management of Data, ACM, N.Y.

[Milankovic]
Milankovic, M. "Update Synchronization in Multiple Database Systems," Ph.D. dissertation, Dept. of EE&CS, University of Massachusetts, Amherst, May 79.

[Minoura 1]
Minoura, T. Stanford University, Stanford, California "A New Concurrency Control Algorithm for Distributed Data Base Systems," Proc. 4th Berkeley Conference on

Distributed Data Management & Computer Networks, August 1979.

[Minoura 2]
Minoura, T. "Maximally Concurrent Transaction Processing," Proc. 3rd Berkeley Workshop on Distributed Databases and Computer Networks, August 1978.

[Montgomery]
Montgomery, W.A. "Robust Concurrency Control for a Distributed Information System", Ph.D. dissertation, Laboratory for Computer Science, MIT, Dec. 1978.

[PBR]
Papadimitriou, C. H., Bernstein, P. A. and Rothnie, J. B., Jr. "Some Computational Problems Related to Database Concurrency Control," Proc. Conf. on Theoretical Computer Science, Waterloo, Ontario, August 1977.

[Papadimitriou]
Papadimitriou, C. H. Serializability of Concurrent Updates, Journal of the ACM, Vol. 26, No. 4, Oct. 1979, pp. 631-653.

[PKW]
Peacock, J. K., Manning, E. and Wong, J. W. "Synchronization of Distributed Simulation Using Broadcast Algorithms," Proc. 4th Berkeley Conference on Distributed Data Management & Computer Networks, August 1979.

[RF]
Rahimi, S.K. and W.R. Frants, "A Posted Update Approach to Concurrency Control in Distributed Database Systems", Proc. First International Conf. on Distributed Computing Systems, IEEE, N.Y., Oct. 1979, pp.632-641.

[RS]
Ramirez, R. J. and Santoro, N. "Distributed Control of Updates in Multiple-copy Data Bases: A Time Optimal Algorithm," Proc. 4th Berkeley Conference on Distributed Data Management & Computer Networks, August 1979.

[Reed]
Reed, D.P. Naming and Synchronization .. a Decentralized Computer System, Ph.D. Thesis, M.I.T. Department of Electrical Engineering, Sept. 1978.

[Ries 1]
Ries, D. "The Effect of Concurrency Control on Database Management System Performance", Ph.D. Dissertation, Computer Science Department, University of California, Berkeley, April 1979.

[Ries 2]
Ries, D. Lawrence Livermore Laboratory, Livermore, California "The Effects of Concurrency Control on the Performance of a Distributed Data Management System," Proc. 4th Berkeley Conference on Distributed Data Management & Computer Networks, August 1979.

[Rosen]
Rosen, E. C. "The Updating Protocol of the ARPANET's New Routing Algorithm: A Case Study in Maintaining Identical Copies of a Changing Distributed Data Base," Proc. 4th Berkeley Conference on Distributed Data Management & Computer Networks, August 1979.

[RSL]
Rosenkrantz, D.J., Stearns, R.E. and Lewis, P.M. "System Level Concurrency Control for Distributed Database Systems", ACM Trans. on Database Systems, Vol. 3, No. 2 (June 1978), pp. 178-198.

[RBFG]
Rothnie, J.B., Bernstein, P.A., Fox, S.A., Goodman, N., Hammer, M.M., Landers, T.A., Reeve, C.L., Shipman, D.W., and Wong, E. "Introduction to a System for Distributed Databases", ACM Trans. on Database System, Vol. 5, No. 1, March 1980.

[RG 1]
Rothnie, J.B. and Goodman, N. "An Overview of the Preliminary Design of SDD-1: A System for Distributed Databases", Proc. 1977 Berkeley Workshop on Distributed Data Management and Computer Networks, May 1977, pp. 39-57.

[RG 2]
Rothnie, J. B., Jr., and Goodman, N. "A Survey of Research and Development in Distributed Database Management," Proc. Third Int. Conf. on Very Large Databases, IEEE, 1977.

[RGM]
Rothnie, J. B., Jr., Goodman, N., and Marill, T. "Database Management in Distributed Networks" in F. F. Kuo (ed.), Protocols and Techniques or Data

Communication Networks, Prentice-Hall, Englewood Cliffs, N.J., 1978.

[Schlageter]
Schlageter, G. "Process Synchronization in Database Systems". TODS 3, 3 (Sept. 1978), 248-271.

[SSW]
Sequin, J., G. Sargeant, and P. Wilnes, "A Majority Consensus Algorithm for the Consistency of Duplicated and Distributed Information", Proc. First International Conf. on Distributed Computing Systems, IEEE, N.Y., Oct. 1979, pp.617-624.

[Selinger]
Selinger, P.G., Private Communications, Nov. 1979.

[SM 1]
Shapiro, R.M. and Millstein, R.E. "Reliability and Fault Recovery in Distributed Processing", Oceans '77 Conference Record, Vol. II, Los Angeles, 1977.

[SM 2]
Shapiro, R.M. and Millstein, R.E. NSW Reliability Plan, Massachusetts Computer Associates, Inc., CA-7701-1411, June 10, 1977.

[SK]
Silberschatz, A. and Z. Kedem", Consistency in Hierarchical Database Systems", Journal of the ACM, Vol. 27, No. 1, Jan. 1980, pp. 72-80.

[SLR]
Stearns, R.E., Lewis, P.M. II and Rosenkrantz, D.J. "Concurrency Controls for Database Systems"; Proceedings of the 17th Annual Symposium on Foundations of Computer Science, IEEE, 1976, pp. 19-32.

[Stonebraker]
Stonebraker, M. "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES, IEEE Transactions on Software Engineering, Vol. SE-5, No. 3, May 1979, pp.188-194.

[SN]
Stonebraker, M. and Neuhold, E. "A Distributed Database Version of INGRES", Proc. 2nd Berkeley Workshop on Distributed Data Management and Computer Networks, May, 1977.

[SRT]
Stucki, M.J., Cox, J.R., Roman, G.C. and Turen, P.N. "Coordinating Concurrent Access in a Distributed Database Architecture," Proc. 4th Workshop on Computer Architecture for Non-Numeric Processing, August 1978.

[Takagi]
Takagi, A. "Concurrent and Reliable Updates of Distributed Databases", M.I.T. Lab. for Computer Science, Request for Comments No. 167, Nov. 1978.

[Thomas 1]
Thomas, R.H. "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases", ACM Trans. on Database Systems, Vol. 4, No. 2, June 1979, pp. 180-209.

[Thomas 2]
Thomas, R.H. "A Solution to the Concurrency Control Problem for Multiple Copy Databases", Proc. 1978 COMPCON Conference., IEEE, N.Y.

[Willcox]
Willcox, D.A., "Optimization of a Relational Algebra Query to a Distributed Database Using Statistical Sampling Methods", Doc #234, Center for Advanced Computation, Univ. of Ill., August, 1977.

[Wong]
Wong, E., "Retrieving Dispersed Data in SDD-1: A System for Distributed Databases", Proc. 1977 Berkeley Workshop on Dist. Data Man. and Comp. Netw., May 1977.

A Partial Index of References

1. Certifiers: Badal 2, Casanova, KR, Papadimitriou

2. Concurrency Control Theory: BCG, BS 1, BS 2, BSW, Casanova, EGLT, KP, Lamport 2, Minoura 2, PBR, Papadimitriou, Schlageter, SK, SLR

3. Performance: G-M 1, G-M 2, GS, Ries 1, Reis 2

4. Reliability:

   General: ABDG, AD, BSS, HS 1, HS 2, LS

   Two-Phase Commit: HS 1, HS 2, LS

5. Timestamp-Ordered Scheduling (T/O)

   General: BGRP, BP, BS 2, BSR, HV, KNTH, Lelann, Lin, SM 1, SM 2, Thomas 1, Thomas 2

   Thomas' Write Rule: JT, Thomas 1, Thomas 2

   Multi-Version Timestamp-Ordering: Montgomery, Reed

   Timestamp and Clock Management: Lamport 1, Thomas 1

6. Two-Phase Locking (2PL)

General: BSW, EGLT, GLPT, Gray, Papadimitriou, Schlageter, SK

Distributed 2PL: MPM, RSL, Stonebraker, Minoura 1

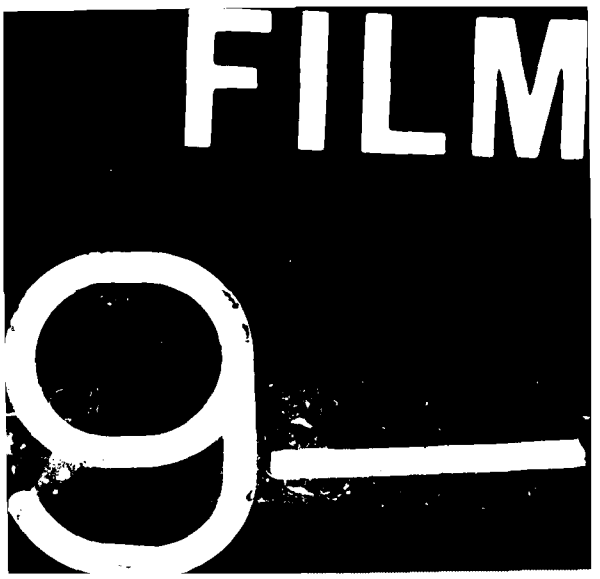
Primary Copy 2PL: Stonebraker, SN

Centralized 2PL: ABDG, AD, G-M 3, G-M 4

Deadlock Detection/Prevention: Gray, KC, KMIT, RSL, Stonebraker

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of technical competence is provided to ESD Program Offices (POs) and other ESD elements. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.


FILM